JBoss Enterprise SOA Platform 5

Smooks ユーザーガイド

このガイドは、ESB パイプラインアクションとして Smooks 変換を使用する開発者を対象としています。

エディッション 5.3.1

Last Updated: 2023-04-30

JBoss Enterprise SOA Platform 5 Smooks ユーザーガイド

このガイドは、ESB パイプラインアクションとして Smooks 変換を使用する開発者を対象としています。
エディッション 5.3.1

David Le Sage
Red Hat Engineering Content Services

Suzanne Dorfield
Red Hat Engineering Content Services

法律上の通知

キーワード

1. SOA 5.3 GA . 2. Enterprise Service Bus . 3. ソフトウェアの操作 . 4. SOA 5.3 GA . 5. Enterprise Service Bus . 6. ソフトウェアの操作 . 7. SOA 5.3 GA . 8. Enterprise Service Bus . 9. JUDDI . 10. 管理タスク . 11. ソフトウェアの操作 . 12. スタートガイド . 13. インストール . 14. Enterprise Service Bus . 15. 管理タスク . 16. SOA 5.3 GA . 17. Enterprise Service Bus . 18. SOA 5.3 GA . 19. Enterprise Service Bus . 20. ソフトウェアの操作 . 21. インストール . 22. Enterprise Service Bus . 23. SOA 5.3 GA . 24. Enterprise Service Bus . 25.

ソフトウェアの操作 . 26. SOA 5.3 GA . 27. インストール . 28. Enterprise Service Bus . 29. ソフトウェアの操作 . 30. SOA 5.3 GA . 31. Enterprise Service Bus . 32. ソフトウェアの操作 . 33. SOA 5.3 GA . 34. Enterprise Service Bus . 35. 管理タスク . 36. SOA 5.3 GA . 37. Enterprise Service Bus . 38. ソフトウェアの操作 . 39. SOA 5.3 GA . 40. Smooks . 41. ソフトウェアの操作 . 42. SOA 5.3 GA . 43. Smooks . 44. ソフトウェアの操作 . 45. スタートガイド . 46. SOA 5.3 GA . 47. インストール . 48. Enterprise Service Bus . 49. 管理タスク . 50. スタートガイド . 51. SOA 5.3 GA . 52. インストール . 53. Enterprise Service Bus . 54. 管理タスク . 55. ソフトウェアの操作 . 56. JDG 6.1 . 57. SOA 5.3 GA . 58. Smooks . 59. ソフトウェアの操作 . 60. SOA 5.3 GA . 61. Smooks . 62. ソフトウェアの操作 . 63. SOA 5.3 GA . 64. Smooks . 65. ソフトウェアの操作 . 66. SOA 5.3 GA . 67. Smooks . 68. ソフトウェアの操作 . 69. SOA 5.3 GA . 70. Smooks . 71. ソフトウェアの操作 . 72. SOA 5.3 GA . 73. Smooks . 74. ソフトウェアの操作 . 75. Smooks . 76. ソフトウェアの操作 . 77. SOA 5.3 GA . 78. Smooks . 79. SOA 5.3 GA . 80. Smooks . 81. ソフトウェアの操作 . 82. SOA 5.3 GA . 83. Smooks . 84. SOA 5.3 GA . 85. Smooks . 86. ソフトウェアの操作 . 87. SOA 5.3 GA . 88. Smooks . 89. ソフトウェアの操作 . 90. SOA 5.3 GA . 91. Smooks . 92. ソフトウェアの操作 . 93. SOA 5.3 GA . 94. Smooks . 95. ソフトウェアの操作 . 96. SOA 5.3 GA . 97. Smooks . 98. ソフトウェアの操作 . 99. SOA 5.3 GA . 100. Smooks . 101. ソフトウェアの操作 . 102. Smooks . 103. ソフトウェアの操作 . 104. SOA 5.3 GA . 105. Smooks . 106. ソフトウェアの操作 . 107. SOA 5.3 GA . 108. Smooks . 109. ソフトウェアの操作 . 110. SOA 5.3 GA . 111. Smooks . 112. ソフトウェアの操作 . 113. SOA 5.3 GA . 114. Smooks . 115. ソフトウェアの操作 . 116. SOA 5.3 GA . 117. Smooks . 118. ソフトウェアの操作 . 119. SOA 5.3 GA . 120. Smooks . 121. ソフトウェアの操作 . 122. SOA 5.3 GA . 123. Smooks . 124. ソフトウェアの操作 . 125. SOA 5.3 GA . 126. Smooks . 127. ソフトウェアの操作 . 128. SOA 5.3 GA . 129. Smooks . 130. ソフトウェアの操作 . 131. SOA 5.3 GA . 132. Smooks . 133. 管理タスク . 134. SOA 5.3 GA . 135. Smooks . 136. ソフトウェアの操作 . 137. SOA 5.3 GA . 138. Smooks . 139. ソフトウェアの操作 . 140. SOA 5.3 GA . 141. Smooks . 142. ソフトウェアの操作 . 143. Smooks . 144. ソフトウェアの操作 . 145. SOA 5.3 GA . 146. Smooks . 147. ソフトウェアの操作 . 148. SOA 5.3 GA . 149. Smooks . 150. ソフトウェアの操作 . 151. SOA 5.3 GA . 152. Smooks . 153. ソフトウェアの操作 . 154. SOA 5.3 GA . 155. Smooks . 156. ソフトウェアの操作 . 157. SOA 5.3 GA . 158. Smooks . 159. ソフトウェアの操作 . 160. SOA 5.3

GA . 161. Smooks . 162. ソフトウェアの操作 . 163. SOA 5.3 GA . 164. Smooks . 165. ソフトウェアの操作 . 166. SOA 5.3 GA . 167. Smooks . 168. ソフトウェアの操作 . 169. SOA 5.3 GA . 170. Smooks . 171. ソフトウェアの操作 . 172. SOA 5.3 GA . 173. Smooks . 174. ソフトウェアの操作 . 175. SOA 5.3 GA . 176. Smooks . 177. ソフトウェアの操作 . 178. SOA 5.3 GA . 179. Smooks . 180. ソフトウェアの操作 . 181. SOA 5.3 GA . 182. Smooks . 183. ソフトウェアの操作 . 184. SOA 5.3 GA . 185. Smooks . 186. ソフトウェアの操作 . 187. SOA 5.3 GA . 188. Smooks . 189. ソフトウェアの操作 . 190. SOA 5.3 GA . 191. Smooks . 192. ソフトウェアの操作 . 193. SOA 5.3 GA . 194. Smooks . 195. ソフトウェアの操作 . 196. SOA 5.3 GA . 197. Smooks . 198. ソフトウェアの操作 . 199. SOA 5.3 GA . 200. Smooks . 201. ソフトウェアの操作 . 202. SOA 5.3 GA . 203. Smooks . 204. ソフトウェアの操作 . 205. SOA 5.3 GA . 206. Smooks . 207. ソフトウェアの操作 . 208. SOA 5.3 GA . 209. Smooks . 210. ソフトウェアの操作 . 211. SOA 5.3 GA . 212. Smooks . 213. ソフトウェアの操作 . 214. SOA 5.3 GA . 215. Smooks . 216. ソフトウェアの操作 . 217. SOA 5.3 GA . 218. Smooks . 219. ソフトウェアの操作 . 220. SOA 5.3 GA . 221. Smooks . 222. ソフトウェアの操作 . 223. SOA 5.3 GA . 224. Smooks . 225. ソフトウェアの操作 . 226. SOA 5.3 GA . 227. Smooks . 228. ソフトウェアの操作 . 229. SOA 5.3 GA . 230. Smooks . 231. ソフトウェアの操作 . 232. SOA 5.3 GA . 233. Smooks . 234. ソフトウェアの操作 . 235. SOA 5.3 GA . 236. Smooks . 237. ソフトウェアの操作 . 238. SOA 5.3 GA . 239. Smooks . 240. ソフトウェアの操作 . 241. SOA 5.3 GA . 242. Smooks . 243. ソフトウェアの操作 . 244. SOA 5.3 GA . 245. Smooks . 246. ソフトウェアの操作 . 247. SOA 5.3 GA . 248. Smooks . 249. ソフトウェアの操作 . 250. SOA 5.3 GA . 251. Smooks . 252. ソフトウェアの操作 . 253. SOA 5.3 GA . 254. Smooks . 255. ソフトウェアの操作 . 256. SOA 5.3 GA . 257. Smooks . 258. ソフトウェアの操作 . 259. SOA 5.3 GA . 260. Smooks . 261. ソフトウェアの操作 . 262. SOA 5.3 GA . 263. Smooks . 264. ソフトウェアの操作 . 265. SOA 5.3 GA . 266. Smooks . 267. ソフトウェアの操作 . 268. SOA 5.3 GA . 269. Smooks . 270. ソフトウェアの操作 . 271. SOA 5.3 GA . 272. Smooks . 273. ソフトウェアの操作 . 274. SOA 5.3 GA . 275. Smooks . 276. ソフトウェアの操作 . 277. SOA 5.3 GA . 278. Smooks . 279. ソフトウェアの操作 . 280. SOA 5.3 GA . 281. Smooks . 282. ソフトウェアの操作 . 283. SOA 5.3 GA . 284. Smooks . 285. ソフトウェアの操作 . 286. SOA 5.3 GA . 287. Smooks . 288. ソフトウェアの操作 . 289. SOA 5.3 GA . 290. Smooks . 291. ソフトウェアの操作 . 292. SOA 5.3 GA . 293. Smooks .

294. ソフトウェアの操作 . 295. SOA 5.3 GA . 296. Smooks . 297. ソフトウェアの操作 . 298. SOA 5.3 GA . 299. Smooks . 300. ソフトウェアの操作 . 301. SOA 5.3 GA . 302. Smooks . 303. ソフトウェアの操作 . 304. SOA 5.3 GA . 305. Smooks . 306. ソフトウェアの操作 . 307. SOA 5.3 GA . 308. Smooks . 309. ソフトウェアの操作 . 310. SOA 5.3 GA . 311. Smooks . 312. ソフトウェアの操作 . 313. SOA 5.3 GA . 314. Smooks . 315. ソフトウェアの操作 . 316. SOA 5.3 GA . 317. Smooks . 318. ソフトウェアの操作 . 319. SOA 5.3 GA . 320. Smooks . 321. ソフトウェアの操作 . 322. SOA 5.3 GA . 323. Smooks . 324. ソフトウェアの操作 . 325. SOA 5.3 GA . 326. Smooks . 327. ソフトウェアの操作 . 328. SOA 5.3 GA . 329. Smooks . 330. ソフトウェアの操作 . 331. SOA 5.3 GA . 332. Smooks . 333. ソフトウェアの操作 . 334. SOA 5.3 GA . 335. Smooks . 336. ソフトウェアの操作 . 337. SOA 5.3 GA . 338. Smooks . 339. ソフトウェアの操作 . 340. SOA 5.3 GA . 341. Smooks . 342. ソフトウェアの操作 . 343. SOA 5.3 GA . 344. Smooks . 345. ソフトウェアの操作 . 346. SOA 5.3 GA . 347. Smooks . 348. ソフトウェアの操作 . 349. SOA 5.3 GA . 350. Smooks . 351. ソフトウェアの操作 . 352. SOA 5.3 GA . 353. Smooks . 354. ソフトウェアの操作 . 355. SOA 5.3 GA . 356. Smooks . 357. ソフトウェアの操作 . 358. SOA 5.3 GA . 359. Smooks . 360. ソフトウェアの操作 . 361. SOA 5.3 GA . 362. Smooks . 363. ソフトウェアの操作 . 364. SOA 5.3 GA . 365. Smooks . 366. ソフトウェアの操作 . 367. SOA 5.3 GA . 368. Smooks . 369. ソフトウェアの操作 . 370. SOA 5.3 GA . 371. Smooks . 372. ソフトウェアの操作 . 373. SOA 5.3 GA . 374. Smooks . 375. ソフトウェアの操作 . 376. SOA 5.3 GA . 377. Smooks . 378. ソフトウェアの操作 . 379. SOA 5.3 GA . 380. Smooks . 381. ソフトウェアの操作 . 382. SOA 5.3 GA . 383. Smooks . 384. ソフトウェアの操作 . 385. SOA 5.3 GA . 386. Smooks . 387. ソフトウェアの操作 . 388. SOA 5.3 GA . 389. Smooks . 390. ソフトウェアの操作 . 391. SOA 5.3 GA . 392. Smooks . 393. ソフトウェアの操作 . 394. SOA 5.3 GA . 395. Smooks . 396. ソフトウェアの操作 . 397. SOA 5.3 GA . 398. Smooks . 399. ソフトウェアの操作 . 400. SOA 5.3 GA . 401. Smooks . 402. ソフトウェアの操作 . 403. SOA 5.3 GA . 404. Smooks . 405. ソフトウェアの操作 . 406. SOA 5.3 GA . 407. Smooks . 408. ソフトウェアの操作 . 409. SOA 5.3 GA . 410. Smooks . 411. ソフトウェアの操作 . 412. SOA 5.3 GA . 413. Smooks . 414. ソフトウェアの操作 . 415. SOA 5.3 GA . 416. Smooks . 417. ソフトウェアの操作 . 418. SOA 5.3 GA . 419. Smooks . 420. ソフトウェアの操作 . 421. SOA 5.3 GA . 422. Smooks . 423. ソフトウェアの操作 . 424. SOA 5.3 GA . 425. Smooks . 426.

ソフトウェアの操作 . 427. SOA 5.3 GA . 428. Smooks . 429. ソフトウェアの操作 . 430. SOA 5.3 GA . 431. Smooks . 432. ソフトウェアの操作 . 433. SOA 5.3 GA . 434. Smooks . 435. ソフトウェアの操作 . 436. SOA 5.3 GA . 437. Smooks . 438. ソフトウェアの操作 . 439. SOA 5.3 GA . 440. Smooks . 441. ソフトウェアの操作 . 442. SOA 5.3 GA . 443. Smooks . 444. ソフトウェアの操作 . 445. SOA 5.3 GA . 446. Smooks . 447. ソフトウェアの操作 . 448. SOA 5.3 GA . 449. Smooks . 450. ソフトウェアの操作 . 451. SOA 5.3 GA . 452. Smooks . 453. ソフトウェアの操作 . 454. SOA 5.3 GA . 455. Smooks . 456. ソフトウェアの操作 . 457. SOA 5.3 GA . 458. Smooks . 459. ソフトウェアの操作 . 460. SOA 5.3 GA . 461. Smooks . 462. ソフトウェアの操作 . 463. SOA 5.3 GA . 464. Smooks . 465. ソフトウェアの操作 . 466. SOA 5.3 GA . 467. Smooks . 468. ソフトウェアの操作 . 469. SOA 5.3 GA . 470. Smooks . 471. ソフトウェアの操作 . 472. SOA 5.3 GA . 473. Smooks . 474. ソフトウェアの操作 . 475. SOA 5.3 GA . 476. Smooks . 477. ソフトウェアの操作 . 478. SOA 5.3 GA . 479. Smooks . 480. ソフトウェアの操作 . 481. SOA 5.3 GA . 482. Smooks . 483. ソフトウェアの操作 . 484. SOA 5.3 GA . 485. Smooks . 486. ソフトウェアの操作 . 487. SOA 5.3 GA . 488. Smooks . 489. ソフトウェアの操作 . 490. SOA 5.3 GA . 491. Smooks . 492. ソフトウェアの操作 . 493. SOA 5.3 GA . 494. Smooks . 495. ソフトウェアの操作 . 496. SOA 5.3 GA . 497. Smooks . 498. ソフトウェアの操作 . 499. SOA 5.3 GA . 500. Smooks . 501. ソフトウェアの操作 . 502. SOA 5.3 GA . 503. Smooks . 504. ソフトウェアの操作 . 505. SOA 5.3 GA . 506. Smooks . 507. ソフトウェアの操作 . 508. SOA 5.3 GA . 509. Smooks . 510. ソフトウェアの操作 . 511. SOA 5.3 GA . 512. Smooks . 513. ソフトウェアの操作 . 514. SOA 5.3 GA . 515. Smooks . 516. ソフトウェアの操作 . 517. SOA 5.3 GA . 518. Smooks . 519. ソフトウェアの操作 . 520. SOA 5.3 GA . 521. Smooks . 522. ソフトウェアの操作 . 523. SOA 5.3 GA . 524. Smooks . 525. ソフトウェアの操作 . 526. SOA 5.3 GA . 527. Smooks . 528. ソフトウェアの操作 . 529. SOA 5.3 GA . 530. Smooks . 531. ソフトウェアの操作 . 532. SOA 5.3 GA . 533. Smooks . 534. ソフトウェアの操作 . 535. アプリケーションサーバー . 536. スタートガイド . 537. インストール . 538. 管理タスク . 539. SOA 5.3 GA . 540. Smooks . 541. ソフトウェアの操作 . 542. SOA 5.3 GA . 543. Smooks . 544. ソフトウェアの操作 . 545. SOA 5.3 GA . 546. Smooks . 547. ソフトウェアの操作 . 548. SOA 5.3 GA . 549. Smooks . 550. ソフトウェアの操作 . 551. SOA 5.3 GA . 552. Smooks . 553. ソフトウェアの操作 . 554. SOA 5.3 GA . 555. Smooks . 556. ソフトウェアの操作 . 557. SOA 5.3

GA . 558. Smooks . 559. ソフトウェアの操作 . 560. SOA 5.3 GA . 561. Smooks . 562. ソフトウェアの操作 . 563. SOA 5.3 GA . 564. Smooks . 565. ソフトウェアの操作 . 566. SOA 5.3 GA . 567. Smooks . 568. ソフトウェアの操作 . 569. SOA 5.3 GA . 570. Smooks . 571. ソフトウェアの操作 . 572. SOA 5.3 GA . 573. Smooks . 574. ソフトウェアの操作 . 575. SOA 5.3 GA . 576. Smooks . 577. ソフトウェアの操作 . 578. SOA 5.3 GA . 579. Smooks . 580. ソフトウェアの操作 . 581. SOA 5.3 GA . 582. Smooks . 583. ソフトウェアの操作 . 584. SOA 5.3 GA . 585. Smooks . 586. ソフトウェアの操作 . 587. SOA 5.3 GA . 588. Smooks . 589. ソフトウェアの操作 . 590. jBPM . 591. SOA 5.3 GA . 592. ソフトウェアの操作 . 593. SOA 5.3 GA . 594. Smooks . 595. ソフトウェアの操作 . 596. SOA 5.3 GA . 597. Smooks . 598. ソフトウェアの操作 . 599. SOA 5.3 GA . 600. Smooks . 601. ソフトウェアの操作 . 602. SOA 5.3 GA . 603. Smooks . 604. ソフトウェアの操作 . 605. SOA 5.3 GA . 606. Smooks . 607. ソフトウェアの操作 . 608. SOA 5.3 GA . 609. Smooks . 610. ソフトウェアの操作 . 611. SOA 5.3 GA . 612. Smooks . 613. ソフトウェアの操作 . 614. SOA 5.3 GA . 615. Smooks . 616. ソフトウェアの操作 . 617. SOA 5.3 GA . 618. Smooks . 619. ソフトウェアの操作 . 620. SOA 5.3 GA . 621. Smooks . 622. ソフトウェアの操作 . 623. SOA 5.3 GA . 624. Smooks . 625. ソフトウェアの操作 . 626. SOA 5.3 GA . 627. Smooks . 628. ソフトウェアの操作 . 629. SOA 5.3 GA . 630. Smooks . 631. ソフトウェアの操作 . 632. SOA 5.3 GA . 633. Smooks . 634. ソフトウェアの操作 . 635. SOA 5.3 GA . 636. Smooks . 637. ソフトウェアの操作 . 638. SOA 5.3 GA . 639. Smooks . 640. ソフトウェアの操作 . 641. SOA 5.3 GA . 642. Smooks . 643. ソフトウェアの操作 . 644. SOA 5.3 GA . 645. Smooks . 646. ソフトウェアの操作 . 647. SOA 5.3 GA . 648. Smooks . 649. ソフトウェアの操作 . 650. SOA 5.3 GA . 651. Smooks . 652. ソフトウェアの操作 . 653. SOA 5.3 GA . 654. Smooks . 655. ソフトウェアの操作 . 656. SOA 5.3 GA . 657. Smooks . 658. ソフトウェアの操作 . 659. SOA 5.3 GA . 660. Smooks . 661. ソフトウェアの操作 . 662. SOA 5.3 GA . 663. Smooks . 664. ソフトウェアの操作 . 665. SOA 5.3 GA . 666. Smooks . 667. ソフトウェアの操作 . 668. SOA 5.3 GA . 669. Smooks . 670. ソフトウェアの操作 . 671. SOA 5.3 GA . 672. Smooks . 673. ソフトウェアの操作 . 674. SOA 5.3 GA . 675. Smooks . 676. ソフトウェアの操作 . 677. SOA 5.3 GA . 678. Smooks . 679. ソフトウェアの操作 . 680. SOA 5.3 GA . 681. Smooks . 682. ソフトウェアの操作 . 683. SOA 5.3 GA . 684. Smooks . 685. ソフトウェアの操作 . 686. SOA 5.3 GA . 687. Smooks . 688. ソフトウェアの操作 . 689. SOA 5.3 GA .

690. Smooks . 691. ソフトウェアの操作 . 692. SOA 5.3 GA . 693. Smooks . 694. ソフトウェアの操作 . 695. SOA 5.3 GA . 696. Smooks . 697. ソフトウェアの操作 . 698. SOA 5.3 GA . 699. Smooks . 700. ソフトウェアの操作 . 701. SOA 5.3 GA . 702. Smooks . 703. ソフトウェアの操作 . 704. SOA 5.3 GA . 705. Smooks . 706. ソフトウェアの操作 . 707. SOA 5.3 GA . 708. Smooks . 709. ソフトウェアの操作 . 710. SOA 5.3 GA . 711. Smooks . 712. ソフトウェアの操作 . 713. SOA 5.3 GA . 714. Smooks . 715. ソフトウェアの操作 . 716. SOA 5.3 GA . 717. Smooks . 718. ソフトウェアの操作 . 719. SOA 5.3 GA . 720. Smooks . 721. ソフトウェアの操作 . 722. SOA 5.3 GA . 723. Smooks . 724. ソフトウェアの操作 . 725. SOA 5.3 GA . 726. Smooks . 727. ソフトウェアの操作 . 728. SOA 5.3 GA . 729. Smooks . 730. ソフトウェアの操作 . 731. SOA 5.3 GA . 732. Smooks . 733. ソフトウェアの操作 . 734. SOA 5.3 GA . 735. Smooks . 736. ソフトウェアの操作 . 737. SOA 5.3 GA . 738. Smooks . 739. ソフトウェアの操作 . 740. SOA 5.3 GA . 741. Smooks . 742. ソフトウェアの操作 . 743. SOA 5.3 GA . 744. Smooks . 745. ソフトウェアの操作 . 746. SOA 5.3 GA . 747. Smooks . 748. ソフトウェアの操作 . 749. SOA 5.3 GA . 750. Smooks . 751. ソフトウェアの操作 . 752. SOA 5.3 GA . 753. Smooks . 754. ソフトウェアの操作 . 755. SOA 5.3 GA . 756. Smooks . 757. ソフトウェアの操作 . 758. SOA 5.3 GA . 759. Smooks . 760. ソフトウェアの操作 . 761. SOA 5.3 GA . 762. Smooks . 763. ソフトウェアの操作 . 764. SOA 5.3 GA . 765. Smooks . 766. ソフトウェアの操作 . 767. SOA 5.3 GA . 768. Smooks . 769. ソフトウェアの操作 . 770. SOA 5.3 GA . 771. Smooks . 772. ソフトウェアの操作 . 773. SOA 5.3 GA . 774. Smooks . 775. ソフトウェアの操作 . 776. SOA 5.3 GA . 777. Smooks . 778. ソフトウェアの操作 . 779. SOA 5.3 GA . 780. Smooks . 781. ソフトウェアの操作 . 782. SOA 5.3 GA . 783. Smooks . 784. ソフトウェアの操作 . 785. SOA 5.3 GA . 786. Smooks . 787. ソフトウェアの操作 . 788. SOA 5.3 GA . 789. Smooks . 790. ソフトウェアの操作 . 791. SOA 5.3 GA . 792. Smooks . 793. ソフトウェアの操作 . 794. SOA 5.3 GA . 795. Smooks . 796. ソフトウェアの操作 . 797. SOA 5.3 GA . 798. Smooks . 799. ソフトウェアの操作 . 800. SOA 5.3 GA . 801. Smooks . 802. ソフトウェアの操作 . 803. SOA 5.3 GA . 804. Smooks . 805. ソフトウェアの操作 .

概要

このドキュメントでは、開発者がメッセージで Smooks 変換を使用する方法について説明します。

目次

はじめに

第1章 はじめに

1.1. ビジネス統合

動的かつアジャイルなビジネスインフラストラクチャーを提供するためには、異なるアプリケーションとデータソースが最小限のオーバーヘッドで相互に通信できるように、サービス指向のアーキテクチャーを用意することが重要です。

JBoss Enterprise SOA Platform は、ビジネスプロセスの変化に合わせて頻繁にプログラムし直す必要なく、ビジネスサービスをオーケストレーションできるフレームワークです。JBoss Enterprise SOA Platform では、ビジネスルールとメッセージの変換およびルーティング機能を使用することで、複数のソースからビジネスデータを操作できます。

バグの報告

1.2. サービス指向アーキテクチャーとは

はじめに

サービス指向アーキテクチャー (SOA) は、1つのプログラムまたはテクノロジーではなく、むしろ、ソフトウェア設計パラダイムと考えてください。

ご存知のように、*ハードウェアバス* は、複数のシステムとサブシステムを結び付ける物理コネクターです。ハードウェアバスを使用する場合は、システムのペア間で多くのポイントツーポイントコネクターを使用する代わりに、各システムを中央バスに接続するだけです。*エンタープライズサービスバス* (ESB: Enterprise Service Bus) は、ソフトウェアでまったく同じことを行います。

ESB は、メッセージングシステムの上にあるアーキテクチャー層に位置します。このメッセージングシステムは、このメッセージングシステムを使用することでサービス間の *非同期通信* を容易にします。実際、ESB を使用している場合、概念的にはすべてが *サービス* (このコンテキストではアプリケーションソフトウェア) またはサービス間で送信される *メッセージ* のいずれかになります。サービスは接続アドレスとして一覧表示されます (*エンドポイント参照* と呼ばれています)。

このコンテキストでは、サービスは必ずしも Web サービスであるとは限らないことに注意することが重要です。File Transfer Protocol や Java Message Service などのトランスポートを使用する他のタイプのアプリケーションもサービスになる可能性があります。

注記

この時点で、Enterprise Service Bus がサービス指向アーキテクチャーと同じものなのかと考えるかもしれません。実は、正確には同じものではありません。 ESB 自体は、サービス指向のアーキテクチャーを形成しません。代わりに、ツールを多く使用して構築できます。特に、SOA で必要な *疎結合* や *非同期メッセージの送信* が容易になります。SOA は、一連の原則、パターン、およびベストプラクティスを指し、単なるソフトウェアではないと考えてください。

バグの報告

1.3. サービス指向アーキテクチャーの重要なポイント

以下は、サービス指向のアーキテクチャーの主要なコンポーネントです。

1. 交換される *メッセージ*
2. サービスリクエスターおよびプロバイダーとして動作する *エージェント*
3. メッセージを送受信できる *共有トランスポートメカニズム*

バグの報告

1.4. JBOSS ENTERPRISE SOA PLATFORM とは

JBoss Enterprise SOA Platform は、エンタープライズアプリケーションインテグレーション (EAI) およびサービス指向アーキテクチャー (SOA) ソリューションを開発するためのフレームワークで、Enterprise Service Bus (JBoss Warehouse) およびビジネスプロセス自動化インフラストラクチャーで構成されます。これを使用すると、ビジネスサービスの構築、デプロイ、統合、オーケストレーションを行うことができます。

バグの報告

1.5. SERVICE-ORIENTED ARCHITECTURE PARADIGM

サービス指向アーキテクチャー (SOA) は、リクエスター、プロバイダー、ブローカーの 3 つのロールで構成されます。

サービスプロバイダー

サービスプロバイダーはサービスへのアクセスを許可し、サービスの説明を作成し、サービスブローカーに公開します。

サービスリクエスター

サービスリクエスターは、サービスブローカーが提供するサービスの説明を検索して、サービスを検出します。リクエスターは、サービスプロバイダーが提供するサービスに対するバインドも行います。

サービスブローカー

サービスブローカーは、サービスの記述のレジストリーをホストします。リクエスターをサービスプロバイダーにリンクします。

バグの報告

1.6. コアおよびコンポーネント

JBoss Enterprise SOA Platform は、データ統合のニーズに対応する包括的なサーバーを提供します。基本的なレベルでは、Enterprise Service Bus によってビジネスルールを更新し、メッセージをルーティングできます。

JBoss Enterprise SOA Platform の中心となるのは、Enterprise Service Bus です。JBoss (ESB) はメッセージを送受信するための環境を作成します。メッセージにアクションを適用して変換し、サービス間でルーティングすることができます。

JBoss Enterprise SOA Platform を構成するコンポーネントは複数あります。ESB に加えて、レジストリー (jUDDI)、変換エンジン (Smooks)、メッセージキュー (HornetQ)、および BPEL エンジン (Riftsaw) があります。

バグの報告

1.7. JBOSS ENTERPRISE SOA PLATFORM のコンポーネント

- 完全な Java EE 準拠のアプリケーションサーバー (JBoss Enterprise Application Platform)
- Enterprise Service Bus (JBoss ESB)
- ビジネスプロセス管理システム (jBPM)
- ビジネスルールエンジン (JBoss ルール)
- オプションの JBoss Enterprise Data Services (EDS) 製品のサポート。

バグの報告

1.8. JBOSS ENTERPRISE SOA PLATFORM の機能

JBoss Enterprise Service Bus (ESB)

ESB はサービス間でメッセージを送信し、異なるタイプのシステムで処理できるようにメッセージを変換します。

Business Process Execution Language (BPEL)

Web サービスを使用して、BPEL 言語でビジネスルールをオーケストレーションできます。BPEL 言語は、ビジネスプロセス命令を簡単に実行するために SOA に含まれています。

Java Universal Description、Discovery and Integration (jUDDI)

これは SOA のデフォルトサービスレジストリーで、ESB 上のサービスに関するすべての情報が格納される場所です。

Smooks

この変換エンジンは SOA と組み合わせて使用してメッセージを処理できます。また、メッセージを分割して正しい宛先に送信するためにも使用できます。

JBoss ルール

これは、SOA にパッケージ化されたルールエンジンです。受信するメッセージからデータを推測して、実行する必要があるアクションを判別できます。

バグの報告

1.9. JBOSS ENTERPRISE SOA PLATFORM の JBOSSESB コンポーネントの機能

JBoss Enterprise SOA Platform の JBossESB コンポーネントは以下をサポートします。

- 複数のトランスポートおよびプロトコル
- リスナーアクションモデル (これにより、サービスを相互に選択可能)
- コンテンツベースのルーティング (JBoss Rules エンジン、XPath、Regex、および Smooks 経由)
- サービスオーケストレーション機能を提供するための JBoss Business Process Manager (jBPM) との統合
- ビジネスルールの開発機能を提供するための JBoss ルールとの統合
- BPEL エンジンとの統合

さらに、ESB を使用すると、レガシーシステムを新しいデプロイメントに統合し、同期または非同期で通信させることができます。

また、Enterprise Service Bus は、以下を可能にするインフラストラクチャーおよびツールセットを提供します。

- さまざまなトランスポート機能 (電子メールや JMS など) で動作するように設定されている。
- 汎用オブジェクトリポジトリーとして使用する。
- プラグ可能なデータ変換機能を実装できる。
- インタラクションのログをサポートする。

重要

ソースコードには、**org.jboss.internal.soa.esb** と **org.jboss.soa.esb** の 2 つのツリーがあります。**org.jboss.internal.soa.esb** パッケージの内容をそのまま使用します。これは、通知なしに変更される可能性があるためです。これとは対照的に、**org.jboss.soa.esb** パッケージ内のすべての内容は、Red Hat の非推奨ポリシーの対象となっています。

バグの報告

1.10. タスク管理

JBoss SOA は、影響を受けるすべてのシステムで汎用的に実行するタスクを指定することにより、タスクを簡素化します。つまり、ユーザーが各ターミナルで個別に実行するようにタスクを設定する必要はありません。ユーザーは、Web サービスを使用してシステムを簡単に接続できます。

JBoss SOA を使用して、マシンごとに複数回ではなく、ネットワーク全体で一度トランザクションをデリゲートすると、時間とコストを削減できます。これにより、エラーが発生する可能性も低くなります。

バグの報告

1.11. 統合のユースケース

ACME Equity は、大規模な金融サービス企業で、多くのデータベースやシステムを所有しています。旧式の COBOL ベースのレガシーシステムや、近年で小規模な企業で取得されるデータベースもあります。ビジネスルールが頻繁に変化するため、これらのデータベースを統合することは困難でコストがかかります。会社は、クライアント向け e-commerce の Web サイトを新たに開発することを希望していますが、現在使用している既存のシステムとは同期しない場合があります。

会社は、安価なソリューションを希望していますが、企業セクターの厳密な規制およびセキュリティー要件に準拠するソリューションを検討しています。会社が望ましくないことは、レガシーデータベースおよびシステムを接続するために glob コードを作成および維持する必要があることです。

JBoss Enterprise SOA Platform は、これらのレガシーシステムを新しいお客様の Web サイトに統合するためにミドルウェアレイヤーとして選択されました。フロントエンドシステムとバックエンドシステムの間のブリッジを提供します。JBoss Enterprise SOA Platform で実装されたビジネスルールは、すぐに簡単に更新できます。

その結果、SOA の統合方法により、古いシステムは新しいシステムと同期できるようになりました。1 カ月あたり数万のトランザクションであっても、ボトルネックはありません。XML、JMS、FTP などのさまざまな統合タイプは、システム間でデータを移動するために使用されます。数多くのエンタープライズ標準のメッセージングシステムの 1 つを JBoss Enterprise SOA Platform にプラグインすることで、柔軟性を高めることができます。

さらに利点は、既存のインフラストラクチャーにより多くのサーバーやデータベースが追加されると、システムを簡単にスケールアップできることです。

バグの報告

1.12. ビジネス環境での JBOSS ENTERPRISE SOA PLATFORM の使用

エラーメッセージが発生する可能性が低いサービスの実装により、コストを削減できます。生産性とソーシングオプションの強化により、継続的なコストを削減できます。

情報およびビジネスプロセスは、接続が増加するため、迅速に共有できます。これは Web サービスによって強化され、クライアントを簡単に接続するために使用できます。

レガシーシステムは Web サービスと組み合わせて使用して、異なるシステムが同じ言語にピークにすることができます。これにより、システムの同期に必要なアップグレードおよびカスタムコードの量が減ります。

バグの報告

第2章 はじめに

2.1. 対象読者

本書は、Smooks とその使用方法を学びたい開発者を対象としています。初心者向けの製品の使用方法の紹介と、高度なプログラミングのトピックが含まれています。

バグの報告

2.2. ガイドの目的

このガイドは、開発者に Smooks、その機能、および実装方法について説明することを目的としています。さまざまなタスク用に Smooks を設定する方法と、拡張タスクを実行する方法を説明します。

バグの報告

2.3. データのバックアップ

警告

Red Hat は、本ガイドに記載されている設定タスクを行う前に、システム設定とデータのバックアップを行うことを推奨します。

バグの報告

2.4. RED HAT ドキュメントサイト

Red Hat の公式ドキュメントサイトは、https://access.redhat.com/knowledge/docs/ です。上記のサイトには、本書を含め、最新版の全ドキュメントが含まれています。

バグの報告

第3章 基本情報

3.1. SMOOKS

Smooks は、フラグメントベースのデータ変換および分析ツールです。これは、メッセージの断片を解釈できる汎用処理ツールです。ビジターロジックを使用してこれを実現します。XSLT または Java で変換ロジックを実装でき、メッセージセットの変換ロジックを一言管理できる管理フレームワークを提供します。

バグの報告

3.2. SMOOKS のビジターロジック

Smooks は *ビジターロジック* を使用します。ビジターは、メッセージの特定のフラグメントに対して特定のアクションを実行する Java コードです。これにより、Smooks はメッセージフラグメントに対してアクションを実行できます。

バグの報告

3.3. メッセージフラグメントの処理

Smooks は、次のタイプのメッセージフラグメント処理をサポートしています。

- **Templating:** XSLT または FreeMarker を使用して、メッセージフラグメントを変換する
- **Java バインド:** バインドメッセージフラグメントデータを Java オブジェクトに
- **Splitting:** メッセージフラグメントを分割し、分割フラグメントを複数のトランスポートと宛先にルーティングします。
- **エンリッチ:** データベースからのデータでメッセージフラグメントをエンリッチします。
- **Persistence:** メッセージフラグメントデータをデータベースに保持します。
- **Validation:** メッセージフラグメントデータに基本的または複雑な検証を実行します。

バグの報告

3.4. 基本処理モデル

以下は、Smooks で実行できるさまざまな変換のリストです。

- XML から XML
- XML to Java
- Java から XML
- Java から Java

- EDI から XML
- EDI から Java
- Java から EDI へ
- CSV から XML

バグの報告

3.5. サポート対象のモデル

Simple API for XML (SAX)

SAX イベントモデルは、XML ソースから生成できる階層 SAX イベントに基づいています。これらには、**startElement**と**endElement** が含まれます。**EDI**、**CSV**、および Java ファイルなど、他の構造化された階層的なデータソースに適用します。

ドキュメントオブジェクトモデル (DOM)

このオブジェクトモデルを使用して、メッセージソースとその最終結果をマッピングします。

注記

最も重要なイベントには、タイトルに **visitBefore** と **visitAfter** が付きます。

バグの報告

3.6. FREEMARKER

FreeMarker はテンプレートエンジンです。これを使用して、**NodeModel** を作成し、テンプレート操作のドメインモデルとして使用できます。Smooks は、この機能にフラグメントベースのテンプレート変換を実行する機能、およびモデルを巨大なメッセージに適用する機能を追加します。

バグの報告

3.7. SAX の使用例

前提条件

- 実装された *SAXVisitor* インターフェイスが必要です。(プロセスのイベントに対応するインターフェイスを選択します。)
- この例では、**ExecutionContext** 名を使用しています。これは、**BoundAttributeStore** クラスを拡張する *パブリックインターフェイス* です。

手順3.1 タスク

1. 新しい Smooks 設定を作成します。これは、<xxx> 要素の **visitBefore** および **visitAfter** イベントでビジターロジックを適用するために使用されます。

2. **visitBefore** および **visitAfter** イベントのロジックをイベントストリーム全体の特定の要素に適用します。ビジターロジックは、<xxx> 要素のイベントに適用されます。

3. **FreeMarker** で Smooks を使用して、巨大なメッセージに XML から XML への変換を実行します。

4. 次のソース形式を挿入します。

```
<order id='332'>
   <header>
      <customer number="123">Joe</customer>
   </header>
   <order-items>
      <order-item id='1'>
         <product>1</product>
         <quantity>2</quantity>
         <price>8.80</price>
      </order-item>
Â
      <!-- etc etc -->
Â
   </order-items>
</order>
```

5. このターゲット形式を挿入します。

```
<salesorder>
   <details>
      <orderid>332</orderid>
      <customer>
         <id>123</id>
         <name>Joe</name>
      </customer>
   <details>
   <itemList>
      <item>
         <id>1</id>
         <productId>1</productId>
         <quantity>2</quantity>
         <price>8.80</price>
      <item>

      <!-- etc etc -->

   </itemList>
</salesorder>
```

6. 次の Smooks 設定を使用します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

   <!--
   Filter the message using the SAX Filter (i.e. not DOM, so no
```

```
    intermediate DOM for the "complete" message - there are "mini" DOMs
    for the NodeModels below)....
    -->
    <params>
        <param name="stream.filter.type">SAX</param>
        <param name="default.serialization.on">false</param>
    </params>

    <!--
    Create 2 NodeModels.  One high level model for the "order"
    (header etc) and then one per "order-item".

    These models are used in the FreeMarker templating resources
    defined below.  You need to make sure you set the selector such
    that the total memory footprint is as low as possible.  In this
    example, the "order" model will contain everything accept the
    <order-item> data (the main bulk of data in the message).  The
    "order-item" model only contains the current <order-item> data
    (i.e. there's max 1 order-item in memory at any one time).
    -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!--
    Apply the first part of the template when we reach the start
    of the <order-items> element.  Apply the second part when we
    reach the end.

    Note the <?TEMPLATE-SPLIT-PI?> Processing Instruction in the
    template.  This tells Smooks where to split the template,
    resulting in the order-items being inserted at this point.
    -->
    <ftl:freemarker applyOnElement="order-items">
        <ftl:template><!--<salesorder>
    <details>
        <orderid>${order.@id}</orderid>
        <customer>
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
    </details>
    <itemList>
        <?TEMPLATE-SPLIT-PI?>
    </itemList>
</salesorder>--></ftl:template>
    </ftl:freemarker>

    <!--
    Output the <order-items> elements.  This will appear in the
    output message where the <?TEMPLATE-SPLIT-PI?> token appears in the
    order-items template.
    -->
    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--        <item>
            <id>${.vars["order-item"].@id}</id>
```

```
            <productId>${.vars["order-item"].product}</productId>
            <quantity>${.vars["order-item"].quantity}</quantity>
            <price>${.vars["order-item"].price}</price>
        </item>
        --></ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

7. このコードを使用して、実行します。

```
Smooks smooks = new Smooks("smooks-config.xml");
try {
    smooks.filterSource(new StreamSource(new FileInputStream("input-message.xml")), new
StreamResult(System.out));
} finally {
    smooks.close();
}
```

8. その結果、XML から XML への変換が行われます。

バグの報告

3.8. カートリッジ

*カートリッジ*は、再利用可能な *コンテンツハンドラー*を含む *Java アーカイブ* (JAR) ファイルです。ほとんどの場合、Smooks 用の大量の Java コードを記述する必要はありません。これは、一部の機能モジュールがカートリッジとして含まれているためです。独自の新しいカートリッジを作成して、smooks-core の基本機能を拡張できます。各カートリッジは、変換プロセスまたは特定の形式の XML 分析をすぐに使用できるサポートを提供します。

バグの報告

3.9. 付属カートリッジ

Smooks に付属のカートリッジは次のとおりです。

- Calc:"milyn-smooks-calc"
- CSV: "milyn-smooks-csv"
- 固定長リーダー: milyn-smooks-fixed-length
- EDI: "milyn-smooks-edi"
- Javabean: milyn-smooks-javabean
- JSON: "milyn-smooks-json"
- ルーティング: milyn-smooks-routing
- テンプレート: milyn-smooks-templating

- CSS: "milyn-smooks-css"
- サーブレット: "milyn-smooks-servlet"
- 保持: milyn-smooks-persistence
- 検証: milyn-smooks-validation

バグの報告

3.10. セレクター

Smooks リソースセレクター はビジターロジックを適用するメッセージフラグメントを Smooks に指示します。これらは、非ビジターロジックの単純なルックアップ値としても機能します。リソースがビジター実装 (**<jb:bean>** や **<ftl:freemarker>** など) の場合、Smooks はリソースセレクターを XPath セレクターとして扱います。リソースには、**Java Binding Resource** と **FreeMarker Template Resource** が含まれます。

バグの報告

3.11. セレクターの使用

セレクターを使用する場合は、次の点が適用されます。

- 設定は、読みやすさのために、厳密に型指定されており、ドメイン固有です。
- 設定は XSD ベースです。これにより、*統合開発環境* を使用する際、オートコンプリートがサポートされます。
- 特定のリソースタイプ (Java バインドの **BeanPopulator** クラスなど) に実際のハンドラーを定義する必要はありません。

バグの報告

3.12. NAMESPACE の宣言

手順3.2 タスク

- コア設定 namespace によって namespace の接頭辞から URI へのマッピングを設定し、次の XML コードを変更して、使用する namespace を含めます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

    <core:namespaces>
        <core:namespace prefix="a" uri="http://a"/>
        <core:namespace prefix="b" uri="http://b"/>
        <core:namespace prefix="c" uri="http://c"/>
        <core:namespace prefix="d" uri="http://d"/>
```

```
    </core:namespaces>

    <resource-config selector="c:item[@c:code = '8655']/d:units[text() = 1]">
        <resource>com.acme.visitors.MyCustomVisitorImpl</resource>
    </resource-config>

</smooks-resource-list>
```

バグの報告

3.13. フィルタリングプロセスの選択

これは、Smooks が *フィルタリングプロセス* を選択する方法です。

- DOM ビジターインターフェイス (**DOMElementVisitor** および **SerializationUnit**) のみが適用されている場合、DOM 処理モデルは自動的に選択されます。
- すべてのビジターリソースが SAX ビジターインターフェイス (**SAXElementVisitor**) のみを使用する場合は、SAX 処理モデルが自動的に選択されます。
- ビジターリソースが DOM と SAX の両方のインターフェイスを使用する場合は、Smooks リソース設定ファイルで SAX を指定しないかぎり、デフォルトでは、DOM 処理モデルが選択されます。(これは、**<core:filterSettings type="SAX" />** を使用すると、実行されます。)

ビジターリソースには、**リーダー** などの *非要素のビジターリソースは* 含まれません。

バグの報告

3.14. SMOOKS 1.3 でフィルタータイプを SAX に設定する例

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

    <core:filterSettings type="SAX" />

</smooks-resource-list>
```

バグの報告

3.15. DOMMODELCREATOR

DomModelCreator は、メッセージフラグメントのモデルを作成するために Smooks で使用できるクラスです。

バグの報告

3.16. DOM モデルと SAX モデルの混合

- *ノードトラバーサル* (つまり、ノード間の情報の送信) と既存のスクリプト/テンプレートエンジンに DOM (ドキュメントオブジェクトモデル) を使用します。
- **DomModelCreator** ビジタークラスを使用して、SAX モデルと DOM モデルを混在させます。このビジターは、SAX フィルタリングで使用すると、ビジットされた要素から DOM フラグメントを構成します。これにより、ストリーミング環境内で DOM ユーティリティーを使用できます。
- 複数のモデルがネストされている場合、外側のモデルに内側のモデルのデータが含まれることはありません (つまり、同じフラグメントが 2 つのモデル内に共存することはありません)。

```
<order id="332">
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>1</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='2'>
            <product>2</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='3'>
            <product>3</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
    </order-items>
</order>
```

バグの報告

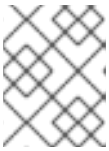
3.17. DOMMODELCREATOR の設定

1. Smooks 内から **DomModelCreator** を設定して、order および注文品目メッセージフラグメントのモデルを作成します。以下の例を参照してください。

```
<resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
</resource-config>
```

2. 次のように、**order** のメモリー内モデルを設定します。

```
<order id='332'>
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items />
</order>
```



注記

新しいモデルごとに前のモデルが上書きされるため、一度に複数の **注文品目** モデルがメモリーに存在することはありません。

バグの報告

3.18. DOMMODELCREATOR に関する詳細情報

- Groovy スクリプト: http://www.smooks.org/mediawiki/index.php?title=V1.3:groovy
- FreeMarker テンプレート: http://www.smooks.org/mediawiki/index.php?title=V1.3:xml-to-xml

バグの報告

3.19. BEAN コンテキスト

Bean コンテキスト には、フィルタリングが発生した際、Smooks がアクセスするオブジェクトが含まれています。実行コンテキストごとに 1 つの Bean コンテキストが作成されます (**Smooks.filterSource** 操作を使用)。カートリッジが作成するすべての Bean は、その beanId に従って、ファイルされます。

バグの報告

3.20. BEAN コンテキストの設定

1. **Smooks.filterSource** プロセスの最後に Bean コンテキストのコンテンツを返すには、**Smooks.filterSource** メソッドの呼び出しで **org.milyn.delivery.java.JavaResult** オブジェクトを指定します。この例は、次の方法を示しています。

```
 //Get the data to filter
StreamSource source = new StreamSource(getClass().getResourceAsStream("data.xml"));

//Create a Smooks instance (cachable)
Smooks smooks = new Smooks("smooks-config.xml");

//Create the JavaResult, which will contain the filter result after filtering
JavaResult result = new JavaResult();

//Filter the data from the source, putting the result into the JavaResult
smooks.filterSource(source, result);

//Getting the Order bean which was created by the Javabean cartridge
Order order = (Order)result.getBean("order");
```

2. 起動時に Bean コンテキストにアクセスするには、これを **BeanContext** オブジェクトで指定します。 **getBeanContext()** メソッドで **ExecutionContext** から取得できます。

3. **BeanContext** からオブジェクトを追加または取得する場合は、まず、**beanIdStore** から **beanId** オブジェクトを取得してください。(**beanId** オブジェクトは、文字列キーもサポートされていますが、文字列キーよりも高いパフォーマンスを保証する特殊なキーです。)

4. **getbeanIdStore()** メソッドを使用して、**ApplicationContext** から **beanIdStore** を取得する必要があります。

5. **beanId** オブジェクトを作成するには、 **register("beanId name")** メソッドを呼び出します。(beanId がすでに登録されていることがわかっている場合は、 **getbeanId("beanId name")** メソッドを呼び出すと、取得できます)。

6. **beanId** オブジェクトは **ApplicationContext** スコープのオブジェクトです。それらをカスタムビジター実装の初期化メソッドに登録してから、ビジターオブジェクトにプロパティーとして配置します。その後、それらを **visitBefore** および **visitAfter** メソッドで使用できます。(**beanId** オブジェクトと **beanIdStore** はスレッドセーフです。)

バグの報告

3.21. プリインストールされた BEAN

次の Bean がプリインストールされています。

- **PUUID**: UniqueId Bean。この Bean は、フィルタリング **ExecutionContext** に一意の識別子を提供します。
- **PTIME**: 時間 Bean。この Bean は、フィルタリング **ExecutionContext** に時間ベースのデータを提供します。

これらの例は、**FreeMarker** テンプレートでこれらの Bean を使用する方法を示しています。

- ExecutionContext の一意の ID (フィルタリングされるメッセージ): **$PUUID.execContext**
- ランダムな一意の ID: **$PUUID.random**
- メッセージフィルタリングの開始時間 (ミリ秒単位): **$PTIME.startMillis**
- メッセージフィルタリングの開始時間 (ナノ秒単位): **$PTIME.startNanos**
- メッセージフィルタリングの開始時間 (日付): **$PTIME.startDate**
- 現在時間 (ミリ秒単位): **$PTIME.nowMillis**
- 現在時間 (ナノ秒単位): **$PTIME.nowNanos**
- 現在時間 (日付): **$PTIME.nowDate**

バグの報告

3.22. 複数の出力/結果

Smooks は次の方法で出力を生成します。

- 結果内インスタンスで。これらは、 **Smooks.filterSource** メソッドに渡された結果インスタンスで返されます。
- フィルタリングプロセス中。これは、フィルタリングプロセス中に生成され、外部エンドポイント (ESB サービス、ファイル、JMS 宛先、データベースなど) に送信される出力によって実現されます。メッセージフラグメントイベントは、外部エンドポイントへの自動ルーティング

をトリガーします。

重要

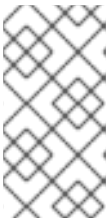
Smooks は、メッセージストリームの 1 つのフィルタリングパスで上記の方法で出力を生成できます。複数の出力を生成するために、メッセージストリームを複数回フィルタリングする必要はありません。

バグの報告

3.23. 結果内インスタンスの作成

- API に見られるように、複数の結果インスタンスを Smooks に提供します。

```
public void filterSource(Source source, Result... results) throws SmooksException
```



注記

Smooks は、同じタイプの複数の結果インスタンスからの結果データの取得をサポートしていません。たとえば、 **Smooks.filterSource** メソッド呼び出しで複数の StreamResult インスタンスを指定できますが、Smooks は、これらの StreamResult インスタンスのうち 1 つ (最初の 1 つ) のみに出力します。

バグの報告

3.24. サポート対象の結果タイプ

Smooks は、標準の **JDK StreamResult** および **DOMResult** の結果タイプだけでなく、次の特殊なタイプでも動作します。

- **JavaResult**: この結果タイプを使用して、Smooks Java Bean コンテキストの内容を取得します。
- **ValidationResult**: この結果タイプを使用して、出力を取得します。
- シンプルな結果タイプ: テストを書く際、これを使用します。これは、**StringWriter** をラップする **StreamResult** 拡張です。

バグの報告

3.25. イベントストリームの結果

Smooks がメッセージを処理すると、イベントのストリームが生成されます。**StreamResult** または **DOMResult** が **Smooks.filterSource** 呼び出しで提供される場合、デフォルトでは、Smooks はイベントストリーム (ソースによって生成される) を提供された結果に XML としてシリアル化します。(シリアル化の前に、ビジターロジックをイベントストリームに適用できます。)

注記

これは、標準の 1 入力/1 XML 出力の文字ベースの変換を実行するために使用される機能です。

バグの報告

3.26. フィルタリングプロセス中

Smooks は、**Smooks.filterSource** プロセス中に、さまざまな出力を生成します。(これは、メッセージの最後に到達する前に、メッセージイベントストリーム中に、発生します。) この例は、他のプロセスによる実行のために、メッセージフラグメントを分割して、さまざまなエンドポイントにルーティングするために使用される場合です。

Smooks は、メッセージデータをバッチ処理して、完全なメッセージをフィルタリングした後、すべての出力を生成することはありません。これは、パフォーマンスが影響を受けるためであり、メッセージイベントストリームを利用して、フラグメント変換およびルーティング操作をトリガーできるためです。大きなメッセージは、プロセスをストリーミングすることによって、送信されます。

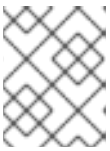
バグの報告

3.27. SMOOKS 実行プロセスの確認

1. Smooks から実行レポートを取得するには、レポートを生成するように、**ExecutionContext** クラスを設定する必要があります。(Smooks は、メッセージを処理する際、イベントを発行します。) 次のサンプルコードは、HTML レポートを生成するように、Smooks を設定する方法を示しています。

```
Smooks smooks = new Smooks("/smooks/smooks-transform-x.xml");
ExecutionContext execContext = smooks.createExecutionContext();

execContext.setEventListener(new HtmlReportGenerator("/tmp/smooks-report.html"));
smooks.filterSource(execContext, new StreamSource(inputStream), new
StreamResult(outputStream));
```

2. **HtmlReportGenerator** 機能を使用すると、デバッグ時に役立ちます。



注記

Web ページ http://www.milyn.org/docs/smooks-report/report.html でサンプルレポートを確認できます。

注記

また、カスタムの **ExecutionEventListener** 実装を作成できます。

バグの報告

3.28. フィルタリングプロセスの終了

1. メッセージの最後に到達する前に、Smooks フィルタリングプロセスを終了するには、<**core:terminate**> 設定を Smooks 設定に追加します。(これは SAX で機能し、DOM では必要ありません。)

   メッセージの顧客フラグメントの最後でフィルタリングを終了する設定例は次のとおりです。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

   <!-- Visitors... -->
   <core:terminate onElement="customer" />

</smooks-resource-list>
```

2. メッセージの先頭で終了するには (**visitBefore**イベント)、次のコードを使用します。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

   <!-- Visitors... -->

   <core:terminate onElement="customer" terminateBefore="true" />

</smooks-resource-list>
```

バグの報告

3.29. グローバル設定

デフォルトプロパティー

デフォルトプロパティーは、**<resource-config>** 属性のデフォルト値を指定します。これらのプロパティーは、対応する **<resource-config>** が属性の値を指定しない場合、**SmooksResourceConfiguration** クラスに自動的に適用されます。

グローバルパラメーター

すべての **<resource-config>** の **<param>** 要素を指定できます。これらのパラメーター値は、**SmooksResourceConfiguration** によって実行時に利用できるか、そうでない場合は、**@ConfigParam** アノテーションによって注入されます。

グローバル設定パラメーターは 1 か所に定義されます。すべてのランタイムコンポーネントは、**ExecutionContext** を使用して、それらにアクセスできます。

バグの報告

3.30. グローバル設定パラメーター

1. 次のように、グローバルパラメーターは、**<params>** 要素で指定されます。

```
<params>
   <param name="xyz.param1">param1-val</param>
</params>
```

2. **ExecutionContext** でグローバルパラメーターにアクセスします。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
   default-selector="order">

   <resource-config>
      <resource>com.acme.VisitorA</resource>
      ...
   </resource-config>

   <resource-config>
      <resource>com.acme.VisitorB</resource>
      ...
   </resource-config>

<smooks-resource-list>
```

バグの報告

3.31. デフォルトプロパティー

デフォルトプロパティーは、Smooks 設定のルート要素に設定でき、**smooks-conf.xml** ファイルのリソース設定に適用されます。すべてのリソース設定のセレクター値が同じである場合は、**default-selector=order** を指定できます。つまり、すべてのリソース設定でセレクターを指定する必要はありません。

バグの報告

3.32. デフォルトプロパティーの設定例

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
   default-selector="order">

   <resource-config>
      <resource>com.acme.VisitorA</resource>
      ...
   </resource-config>

   <resource-config>
      <resource>com.acme.VisitorB</resource>
      ...
   </resource-config>

<smooks-resource-list>
```

バグの報告

3.33. デフォルトプロパティーのオプション

default-selector

他のセレクターが定義されていない場合、これは、Smooks 設定ファイル内のすべての resource-config 要素に適用されます。

default-selector-namespace

これはデフォルトのセレクター namespace です。他の namespace が定義されていない場合に使用されます。

default-target-profile

これはデフォルトのターゲットプロファイルです。他のターゲットプロファイルが定義されていない場合、Smooks 設定ファイル内のすべてのリソースに適用されます。

default-condition-ref

これは、条件識別子によってグローバル条件を参照します。この条件は、グローバルに定義された条件を参照しない空の条件要素 (つまり、<condition/>) を定義するリソースに適用されます。

バグの報告

3.34. フィルター設定

- フィルタリングオプションを設定するには、smooks-core 設定 namespace を使用します。以下の例を参照してください。

```
;smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">
   <core:filterSettings type="SAX" defaultSerialization="true"
      terminateOnException="true" readerPoolSize="3" closeSource="true"
      closeResult="true" rewriteEntities="true" />

      .. Other visitor configs etc...

</smooks-resource-list>
```

バグの報告

3.35. フィルターオプション

type

これにより、SAX または DOM のいずれかから使用される処理モデルのタイプが決まります。(デフォルトは DOM です。)

defaultSerialization

これは、デフォルトのシリアル化をオンにするかどうかを決定します。デフォルト値は **true** です。

これをオンにすると、**Smooks.filterSource** メソッドに提供された結果オブジェクトの **StreamResult** (または **DOMResult**) を探し、デフォルトでは、その結果に対するすべてのイベントをシリアル化するように、Smooks に指示します。

グローバル設定パラメーターを使用して、この動作をオフにするか、結果ライターの所有権を取得する (SAX フィルタリングを使用する場合) か、DOM を変更する (DOM フィルタリングを使用する場合) フラグメントのビジター実装をターゲットにして、フラグメント単位でこの動作をオーバーライドできます。

terminateOnException

これを使用して、例外によって処理を終了する必要があるかどうかを判断します。デフォルト設定は **true** です。

closeSource

これにより、**Smooks.filterSource** メソッドに渡されたソースインスタンスストリームが閉じます (デフォルトは **true**)。ここでの例外は、まったく閉じられない **System.in** です。

closeResult

これにより、**Smooks.filterSource** メソッドに渡された結果ストリームが閉じます (デフォルトは **true**)。ここでの例外は、まったく閉じない **System.out** と **System.err** です。

rewriteEntities

これを使用して、XML の読み取りおよび書き込み (デフォルトのシリアル化) 時に XML エンティティーを書き換えます。

readerPoolSize

これにより、リーダープールのサイズが設定されます。一部のリーダーの実装は、作成に非常にコストがかかります。リーダーインスタンスをプールする (つまり、それらを再利用する) と、特に多くの小さなメッセージを処理する場合は、パフォーマンスが大幅に向上する可能性があります。この設定のデフォルト値は **0** です (つまり、プールされません。メッセージごとに新しいリーダーインスタンスが作成されます)。

アプリケーションのスレッドモデルに合わせて、これを設定します。

バグの報告

第4章 入力データの消費

4.1. ストリームリーダー

*ストリームリーダー*は、**XMLReader** インターフェイス (または **SmooksXMLReader** インターフェイス) を実装するクラスです。Smooks は、ストリームリーダーを使用して、ソースメッセージデータストリームから SAX イベントのストリームを生成します。 **XMLReaderFactory.createXMLReader()** はデフォルトの XMLReader です。特殊な XML リーダーを設定することにより、非 XML データソースを読み取るように設定できます。

バグの報告

4.2. XMLREADER の設定

これは、ハンドラー、機能、およびパラメーターを使用するように、XML を設定する方法の例です。

```
<reader class="com.acme.ZZZZReader">
    <handlers>
        <handler class="com.X" />
        <handler class="com.Y" />
    </handlers>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
    <params>
        <param name="param1">val1</param>
        <param name="param2">val2</param>
    </params>
</reader>
```

バグの報告

4.3. XML リーダーの機能の設定

- デフォルトでは、Smooks は XML データを読み取ります。デフォルトの XML リーダーに機能を設定するには、設定からクラス名を省略します。

```
<reader>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
</reader>
```

バグの報告

4.4. CSV リーダーの設定

1. リーダーを設定するには、http://www.milyn.org/xsd/smooks/csv-1.2.xsd 設定 namespace を使用します。

   基本的な設定は次のとおりです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <!--
    Configure the CSV to parse the message into a stream of SAX events.
    -->
    <csv:reader fields="firstname,lastname,gender,age,country" separator="|" quote="'"
skipLines="1" />

</smooks-resource-list>
```

2. 次のイベントストリームが表示されます。

```
<csv-set>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
</csv-set>
```

バグの報告

4.5. 設定の定義

1. XML 設定でフィールドを定義するには、 fields 属性で名前のコンマ区切りリストを使用する必要があります。

2. フィールド名が XML 要素名と同じ命名規則に従っていることを確認してください。

3. - 文字、数字、その他の文字を含めることができる
   - 数字または句読点で始めることはできない

- 文字 xml (または XML や Xml など) で開始することはできない
- スペースを含めることはできない

4. csv-set および csv-record 要素名を変更できるように、rootElementName および recordElementName 属性を設定します。これらの名前にも同じルールが適用されます。

5. フィールドごとに文字列操作関数を定義できます。これらの関数は、データが SAX イベントに変換される前に、実行されます。フィールド名の後に定義し、2 つを疑問符で区切ります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="lastname?trim.capitalize,country?upper_case" />

</smooks-resource-list>
```

6. Smooks が CSV レコードのフィールドを無視するようにするには、フィールドの設定値として $ignore$ トークンを指定する必要があります。$ignore$ トークンの後に数字を続けるだけで、無視するフィールドの数を指定します (したがって、**$ignore$3** を使用して、次の 3 つのフィールドを無視します)。 **$ignore$+** を使用して、CSV レコードの最後まで、すべてのフィールドを無視します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,$ignore$2,age,$ignore$+" />

</smooks-resource-list>
```

バグの報告

4.6. CSV レコードを JAVA オブジェクトにバインドする

1. CSV レコードを Java オブジェクトに変換する方法については、以下をお読みください。この例では、CSV レコードを個人に使用します。

```
Tom,Fennelly,Male,4,Ireland
Mike,Fennelly,Male,2,Ireland
```

2. このコードを入力して、レコードを個人にバインドします。

```
public class Person {
    private String firstname;
    private String lastname;
```

```
    private String country;
    private Gender gender;
    private int age;
}

public enum Gender {
    Male,
    Female;
}
```

3. 次のコードを入力し、タスクに合わせて、変更します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <!-- Note how the field names match the property names on the Person class. -->
        <csv:listBinding beanId="people" class="org.milyn.csv.Person" />
    </csv:reader>

</smooks-resource-list>
```

4. 設定を実行するには、次のコードを使用します。

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

5. CSV レコードセットからマップを作成できます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <csv:mapBinding beanId="people" class="org.milyn.csv.Person" keyField="firstname" />
    </csv:reader>

</smooks-resource-list>
```

6. 上記の設定では、個人ごとの firstname 値に応じた個人インスタンスのマップが生成されます。実行方法は次のとおりです。

```
Smooks smooks = new Smooks(configStream);
```

```
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

Map<String, Person> people = (Map<String, Person>) result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Mike");
```

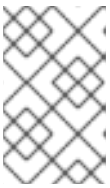
仮想モデルもサポートされているため、class 属性を **java.util.Map** として定義し、CSV フィールド値をバインドして、インスタンスをマッピングし、リストまたはマップに追加できます。

バグの報告

4.7. レコードセットの CSV リーダーの設定

1. CSV リーダーを使用して、Smooks インスタンスを設定し、個人レコードセットを読み取るには、以下のコードを使用します。レコードを個人インスタンスのリストにバインドします。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
CSVReaderConfigurator("firstname,lastname,gender,age,country")
                .setBinding(new CSVBinding("people", Person.class, CSVBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(csvReader), result);

List<Person> people = (List<Person>) result.getBean("people");
```



注記

オプションで Java Bean を設定することもできます。代わりに (または追加で)、他のビジター実装を使用して、CSV レコードセットを処理するように、Smooks インスタンスをプログラムで設定できます。

2. CSV レコードのデータを反映する Java 型のリストまたはマップに CSV のレコードをバインドするには、**CSVListBinder** または **CSVMapBinder** クラスを使用します。

```
// Note: The binder instance should be cached and reused...
CSVListBinder binder = new CSVListBinder("firstname,lastname,gender,age,country",
Person.class);

List<Person> people = binder.bind(csvStream);

CSVMapBinder:

// Note: The binder instance should be cached and reused...
CSVMapBinder binder = new CSVMapBinder("firstname,lastname,gender,age,country",
```

```
Person.class, "firstname");

Map<String, Person> people = binder.bind(csvStream);
```

バインドプロセスをさらに制御する必要がある場合は、下位レベルの API の使用に戻ります。

バグの報告

4.8. 固定長リーダーの設定

1. 固定長リーダーを設定するには、次のように、http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd 設定 namespace を変更します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <!--
    Configure the Fixed length to parse the message into a stream of SAX events.
    -->
    <fl:reader fields="firstname[10],lastname[10],gender[1],age[2],country[2]" skipLines="1" />

</smooks-resource-list>
```

入力ファイルの例は次のとおりです。

```
#HEADER
Tom       Fennelly  M 21 IE
Maurice  Zeijen     M 27 NL
```

生成されるイベントストリームは次のとおりです。

```
<set>
    <record>
        <firstname>Tom       </firstname>
        <lastname>Fennelly  </lastname>
        <gender>M</gender>
        <age> 21</age>
        <country>IE</country>
    </record>
    <record>
        <firstname>Maurice  </firstname>
        <lastname>Zeijen     </lastname>
        <gender>M</gender>
        <age>27</age>
        <country>NL</country>
    </record>
</set>
]]>
```

2. 次のように、文字列操作関数を定義します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <!--
    Configure the fixed length reader to parse the message into a stream of SAX events.
    -->
    <fl:reader fields="firstname[10]?
trim,lastname[10]trim.capitalize,gender[1],age[2],country[2]" skipLines="1" />

</smooks-resource-list>
```

3. 選択した場合は、これらのフィールドを無視することもできます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <fl:reader fields="firstname,$ignore$[2],age,$ignore$[10]" />

</smooks-resource-list>
```

バグの報告

4.9. 固定長レコードの設定

1. 固定長レコードを個人にバインドするには、以下の設定を参照してください。この例では、次のサンプルレコードを使用します。

```
Tom       Fennelly  M 21 IE
Maurice  Zeijen     M 27 NL
```

これは、個人にそれらをバインドする方法です。

```
[
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private String gender;
    private int age;
}
```

2. 次のように、レコードを設定します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <fl:reader fields="firstname[10]?trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]">
        <!-- Note how the field names match the property names on the Person class. -->
        <fl:listBinding BeanId="people" class="org.milyn.fixedlength.Person" />
    </fl:reader>

</smooks-resource-list>
```

3. 次のように、実行します。

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

4. オプションで、この設定を使用して、固定長レコードセットからマップを作成します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd">

    <fl:reader fields="firstname[10]?trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]">
        <fl:mapBinding BeanId="people" class="org.milyn.fixedlength.Person"
keyField="firstname" />
    </fl:reader>
```

5. これは、生成された個人インスタンスのマップを実行する方法です。

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

Map<String, Person> people = (Map<String, Person>) result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Maurice");
```

仮想モデルもサポートされているため、class 属性を java.util.Map として定義し、固定長フィールド値をバインドして、インスタンスをマッピングし、リストまたはマップに追加することができます。

バグの報告

4.10. プログラムによる固定長リーダーの設定

1. 次のコードを使用して、個人レコードセットを読み取るように、固定長リーダーを設定し、レコードセットを個人インスタンスのリストにバインドします。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new FixedLengthReaderConfigurator("firstname[10]?
trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]")
                .setBinding(new FixedLengthBinding("people", Person.class,
FixedLengthBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

   Java バインドの設定は必須ではありません。代わりに、他のビジター実装を使用して、固定長レコードセットにさまざまな形式の処理を実行するように、Smooks インスタンスをプログラムで設定できます。

2. 固定長レコードを、固定長レコードのデータを反映する Java 型のリストまたはマップに直接バインドするには、FixedLengthListBinder クラスまたは FixedLengthMapBinder クラスを使用します。

```
// Note: The binder instance should be cached and reused...
FixedLengthListBinder binder = new FixedLengthListBinder("firstname[10]?
trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]", Person.class);

List<Person> people = binder.bind(fixedLengthStream);

FixedLengthMapBinder:

// Note: The binder instance should be cached and reused...
FixedLengthMapBinder binder = new FixedLengthMapBinder("firstname[10]?
trim,lastname[10]?trim,gender[1],age[3]?trim,country[2]", Person.class, "firstname");

Map<String, Person> people = binder.bind(fixedLengthStream);
```

   バインドプロセスをさらに制御する必要がある場合は、下位レベルの API に戻ります。

バグの報告

4.11. EDI 処理

1. Smooks で EDI 処理を利用するには、http://www.milyn.org/xsd/smooks/edi-1.2.xsd 設定 namespace にアクセスします。

2. ニーズに合わせて、この設定を変更します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:edi="http://www.milyn.org/xsd/smooks/edi-1.2.xsd">
```

```
<!--
   Configure the EDI Reader to parse the message stream into a stream of SAX events.
   -->
   <edi:reader mappingModel="edi-to-xml-order-mapping.xml" validate="false"/>
</smooks-resource-list>
```

バグの報告

4.12. EDI 処理条件

- mappingModel: これは、EDI メッセージストリームを Smooks で処理できる SAX イベントのストリームに変換するための EDI マッピングモデル設定を定義します。
- validate: この属性は、EDI パーサーでのデータ型の検証をオンまたはオフにします。(デフォルトでは、検証はオンになっています。) 冗長性を避けるために、EDI データが Java オブジェクトモデルにバインドされている場合は、EDI リーダーでデータ型の検証をオフにします (jb:bean のような Java バインドを使用)。

バグの報告

4.13. EDI から SAX

EDI から SAX へのイベントマッピングプロセスは、EDI リーダーに提供されるマッピングモデルに基づいています。(このモデルは常に http://www.milyn.org/xsd/smooks/edi-1.2.xsd スキーマを使用する必要があります。このスキーマから、グループ内のグループ、繰り返しセグメント、繰り返しセグメントグループなど、セグメントグループがサポートされていることがわかります。)

medi:segment 要素は、2 つのオプションの属性 minOccurs と maxOccurs をサポートします。(いずれの場合も、デフォルト値は 1 です。) これらの属性を使用して、セグメントの特性を制御します。-1 の maxOccurs 値は、セグメントが (バインドされていない) EDI メッセージのその場所で何度でも繰り返すことができることを示します。

セグメントグループを追加するには、segmentGroup 要素を使用します。セグメントグループは、グループ内の最初のセグメントに一致します。ネストされた segmentGroup 要素を含めることができますが、segmentGroup の最初の要素はセグメントである必要があります。segmentGroup 要素は minOccurs および maxOccurs *カーディナリティ* をサポートします。また、オプションの xmlTag 属性をサポートします。これが存在する場合は、一致したセグメントグループによって生成された XML が、xmlTag 属性値の名前を持つ要素に挿入されます。

バグの報告

4.14. EDI から SAX へのイベントマッピング

EDI を SAX イベントにマッピングする場合、セグメントは次のいずれかの方法で照合されます。

- セグメントコード (segcode) の完全一致。
- segcode 属性が正規表現パターンを定義するフルセグメントの正規表現パターンマッチ (たとえば、**segcode="1A*a.*"**)。

- 必須: フィールド、コンポーネント、およびサブコンポーネントの設定は、そのフィールド、コンポーネント、またはサブコンポーネントに値が必要であるというフラグを立てる必須の属性をサポートします。
- デフォルトでは、値は必要ありません (フィールド、コンポーネント、およびサブコンポーネント)。
- truncatable: セグメント、フィールド、およびコンポーネント設定は、truncatable 属性をサポートします。セグメントの場合は、つまり、そのセグメントが必須ではない後続フィールドを指定していない場合、パーサーエラーは生成されません (上記の必須の属性を参照)。フィールド/コンポーネントおよびコンポーネント/サブコンポーネントについても同様です。
- デフォルトでは、セグメント、フィールド、およびコンポーネントは切り捨てられません。

したがって、フィールド、コンポーネント、またはサブコンポーネントは、次のいずれかの状態でメッセージに存在できます。

- 値とともに存在する (**required="true"**)
- 値なしで存在する (**required="false"**)
- 存在しない (**required="false" and truncatable="true"**)

バグの報告

4.15. セグメント定義

セグメント定義を再利用できます。以下は、インポート機能を示す設定です。

```
<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-1.2.xsd">

    <medi:import truncatableSegments="true" truncatableFields="true" truncatableComponents="true"
resource="example/edi-segment-definition.xml" namespace="def"/>

    <medi:description name="DVD Order" version="1.0"/>

    <medi:delimiters segment="
" field="*" component="^" sub-component="~" escape="?"/>

    <medi:segments xmltag="Order">
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:HDR" segcode="HDR"
xmltag="header"/>
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:CUS" segcode="CUS"
xmltag="customer-details"/>
        <medi:segment minOccurs="0" maxOccurs="-1" segref="def:ORD" segcode="ORD"
xmltag="order-item"/>
    </medi:segments>

</medi:edimap>
```

バグの報告

4.16. セグメント用語

セグメントおよび子セグメントを含むセグメントは、今後の再利用を容易にするために、別のファイルに分割できます。

- segref: これには、インポートするセグメントを参照する namespace:name が含まれます。
- truncatableSegments: これは、インポートされたリソースマッピングファイルで指定された truncatableSegments をオーバーライドします。
- truncatableFields: これは、インポートされたリソースマッピングファイルで指定された truncatableFields をオーバーライドします。
- truncatableComponents: これは、インポートされたリソースマッピングファイルで指定された truncatableComponents をオーバーライドします。

バグの報告

4.17. TYPE 属性

次の例は、type 属性のサポートを示しています。

```
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-1.2.xsd">

    <medi:description name="Segment Definition DVD Order" version="1.0"/>

    <medi:delimiters segment="
" field="*" component="^" sub-component="~" escape="?"/>

    <medi:segments xmltag="Order">

        <medi:segment segcode="HDR" xmltag="header">
            <medi:field xmltag="order-id"/>
            <medi:field xmltag="status-code" type="Integer"/>
            <medi:field xmltag="net-amount" type="BigDecimal"/>
            <medi:field xmltag="total-amount" type="BigDecimal"/>
            <medi:field xmltag="tax" type="BigDecimal"/>
            <medi:field xmltag="date" type="Date" typeParameters="format=yyyyHHmm"/>
        </medi:segment>

    </medi:segments>

</medi:edimap>
```

型システムは、次のようなさまざまなことに使用できます。

- フィールド検証
- Edifact Java コンパイル

バグの報告

4.18. EDIREADERCONFIGURATOR

- 次のコードのように、**EDIReaderConfigurator** を使用して、EDIReader を使用するように、Smooks インスタンスをプログラムで設定します。

```
Smooks smooks = new Smooks();

// Create and initialise the Smooks config for the parser...
smooks.setReaderConfig(new EDIReaderConfigurator("/edi/models/invoice.xml"));

// Use the smooks as normal
smooks.filterSource(....);
```

バグの報告

4.19. EDIFACT JAVA コンパイラー

Edifact Java コンパイラーは、EDI から Java への移行プロセスを簡素化します。以下を生成します。

- 特定の EDI マッピングモデルの Java オブジェクトモデル。
- EDI マッピングモデルによって記述された EDI メッセージのインスタンスから Java オブジェクトモデルを取り込むための Smooks Java バインド設定。
- **Edifact Java コンパイラー** を使用して、EDI データを Java オブジェクトモデルにバインドするためのファクトリークラス。

バグの報告

4.20. EDIFACT JAVA コンパイラーの例

Edifact Java コンパイラー を使用すると、次のような単純な Java コードを記述できます。

```
// Create an instance of the EJC generated Factory class.  This should normally be cached and
reused...
OrderFactory orderFactory = OrderFactory.getInstance();

// Bind the EDI message stream data into the EJC generated Order model...
Order order = orderFactory.fromEDI(ediStream);

// Process the order data...
Header header = order.getHeader();
Name name = header.getCustomerDetails().getName();
List<OrderItem> orderItems = order.getOrderItems();
```

バグの報告

4.21. EDIFACT JAVA コンパイラーの実行

- **Maven** によって **Edifact Java Compiler** を実行するには、POM ファイルにプラグインを追加します。

```
<build>
   <plugins>
      <plugin>
         <groupId>org.milyn</groupId>
         <artifactId>maven-ejc-plugin</artifactId>
         <version>1.2</version>
         <configuration>
            <ediMappingFile>edi-model.xml</ediMappingFile>
            <packageName>com.acme.order.model</packageName>
         </configuration>
         <executions>
            <execution><goals><goal>generate</goal></goals></execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

バグの報告

4.22. EDIFACT JAVA コンパイラーの MAVEN プラグインパラメーター

プラグインには、次の 3 つの設定パラメーターがあります。

- ediMappingFile: **Maven** プロジェクト内の EDI マッピングモデルファイルへのパス。(オプションです。デフォルトは **src/main/resources/edi-model.xml** です。)
- packageName: 生成された Java アーティファクトが配置される Java パッケージ (Java オブジェクトモデルとファクトリークラス)。
- destDir: 生成されたアーティファクトが作成およびコンパイルされるディレクトリー。(オプションです。デフォルトは **target/ejc** です。)

バグの報告

4.23. ANT による EDIFACT JAVA コンパイラーの実行

- 次のように、EJC タスクを作成および実行します。

```
<target name="ejc">

   <taskdef resource="org/milyn/ejc/ant/anttasks.properties">
      <classpath><fileset dir="/smooks-1.2/lib" includes="*.jar"/></classpath>
   </taskdef>

   <ejc edimappingmodel="src/main/resources/edi-model.xml"
        destdir="src/main/java"
        packagename="com.acme.order.model"/>
```

```
    <!-- Ant as usual from here on... compile and jar the source... -->

</target>
```

バグの報告

4.24. UN/EDIFACT メッセージ交換

Edifact Java コンパイラー の詳細を知る最も簡単な方法は、EJC の例である UN/EDIFACT を確認することです。

Smooks は、次の方法で、UN/EDIFACT メッセージ交換のすぐに使えるサポートを提供します。

- 公式の UN/EDIFACT メッセージ定義 ZIP ディレクトリーから生成された、事前生成された EDI マッピングモデル。これらを使用すると、UN/EDIFACT メッセージ交換をより使いやすい XML 形式に変換できます。
- 純粋な Java を使用して、UN/EDIFACT 交換を簡単に読み書きできるように、事前に生成された Java バインド

バグの報告

4.25. EDI:READER で UN/EDIFACT 交換を使用する

- 次のように、http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd namespace を設定します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:unedifact="http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd">

    <unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d03b-mapping:v1.4"
ignoreNewLines="true" />

</smooks-resource-list>
```

mappingModel 属性は、読者が使用するマッピングモデル ZIP セットの **Maven** アーティファクトを参照する *URN* を定義します。

バグの報告

4.26. UN/EDIFACT 交換を使用するように、SMOOKS を設定する

1. UN/EDIFACT 交換を使用するように、Smooks をプログラムで設定するには (たとえば、UNEdifactReaderConfigurator によって)、次のコードを使用します。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
UNEdifactReaderConfigurator("urn:org.milyn.edi.unedifact:d03b-mapping:v1.4"));
```

2. 含まれているアプリケーションのクラスパスに次を挿入します。
   - 必要な EDI マッピングモデル
   - Smooks EDI カートリッジ
3. 設定に必要な複数の **Maven** 依存関係がある場合があります。以下の例を参照してください。

```
<dependency>
   <groupId>org.milyn</groupId>
   <artifactId>milyn-smooks-edi</artifactId>
   <version>1.4</version>
</dependency>

<!-- Required Mapping Models -->
<dependency>
   <groupId>org.milyn.edi.unedifact</groupId>
   <artifactId>d93a-mapping</artifactId>
   <version>v1.4</version>
</dependency>
<dependency>
   <groupId>org.milyn.edi.unedifact</groupId>
   <artifactId>d03b-mapping</artifactId>
   <version>v1.4</version>
</dependency>
```

4. アプリケーションがクラスパスに EDI マッピングモデルの ZIP セットを追加したら、unedifact:reader 設定の mappingModel 属性値として URN を使用して **Maven** アーティファクトを参照するだけで、このモデルを使用するように、**Smooks** を設定できます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:unedifact="http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd"

   <unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d03b-mapping:v1.4"
ignoreNewLines="true" />

</smooks-resource-list>
```

バグの報告

4.27. MAPPINGMODEL

mappingModel 属性は、コンマで区切られた複数の EDI Mapping Models URN を定義できます。そうすると、異なるディレクトリーに定義された複数の UN/EDIFACT メッセージを処理する UN/EDIFACT リーダープロセスの交換が容易になります。

マッピングモデルの ZIP セットは、すべての UN/EDIFACT ディレクトリーに使用できます。これらを **Maven SNAPSHOT** および **Central** リポジトリーから取得し、標準の **Maven** 依存関係管理を使用して、アプリケーションに追加します。

バグの報告

4.28. MAPPINGMODEL の設定

1. D93A マッピングモデル ZIP セットをアプリケーションクラスパスに追加するには、アプリケーションの POM ファイルに次の依存関係を設定します。

```
<!-- The mapping model sip set for the D93A directory... -->
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d93a-mapping</artifactId>
    <version>v1.4</version>
</dependency>
```

2. 次のように、**unedifact:reader** 設定を Smooks 設定に追加して、この ZIP セットを使用するように、Smooks を設定します。

```
<unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d93a-mapping:v1.4" />
```

Maven アーティファクトの依存情報から派生した URN を使用して、リーダーを設定する方法に注意してください。

3. 複数のマッピングモデル ZIP セットをアプリケーションのクラスパスに追加することもできます。これを行うには、URN をコンマで区切って、**unedifact:reader** 設定にそのすべてを追加します。

4. ツールには、事前生成された Java バインドモデルセットが用意されています (マッピングモデルの ZIP セットごとに 1 つ存在します)。これらを使用して、生成された非常に単純なファクトリークラスを使用して、UN/EDIFACT 交換を処理します。

バグの報告

4.29. D03B UN/EDIFACT メッセージ交換の処理

1. D03B UN/EDIFACT メッセージ交換を処理するには、次の例に従います。

```
Reading:

// Create an instance of the EJC generated factory class... cache this and reuse !!!
D03BInterchangeFactory factory = D03BInterchangeFactory.getInstance();

// Deserialize the UN/EDIFACT interchange stream to Java...
UNEdifactInterchange interchange = factory.fromUNEdifact(ediInStream);

// Need to test which interchange syntax version.  Supports v4.1 at the moment...
if(interchange instanceof UNEdifactInterchange41) {
    UNEdifactInterchange41 interchange41 = (UNEdifactInterchange41) interchange;

    for(UNEdifactMessage41 message : interchange41.getMessages()) {
        // Process the messages...

        Object messageObj = message.getMessage();

        if(messageObj instanceof Invoic) {
```

```
            // It's an INVOIC message....
            Invoic invoic = (Invoic) messageObj;
            ItemDescription itemDescription = invoic.getItemDescription();
            // etc etc....
        } else if(messageObj instanceof Cuscar) {
            // It's a CUSCAR message...
        } else if(etc etc etc...) {
            // etc etc etc...
        }
    }
}


Writing:

factory.toUNEdifact(interchange, ediOutStream);
```

2. **Maven** を使用して、そのディレクトリーのバインド依存関係を追加することにより、D03B メッセージ交換を処理する機能を追加します (**Maven SNAPSHOT** および **Central** リポジトリーによって配布される、事前に生成された UN/EDIFACT Java オブジェクトモデルを使用することもできます)。

```
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d03b-binding</artifactId>
    <version>v1.4</version>
</dependency>
```

バグの報告

4.30. JSON データの処理

1. JSON データを処理するには、JSON リーダーを設定する必要があります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader/>

</smooks-resource-list>
```

2. 次の設定オプションを使用して、ルート、ドキュメント、および配列要素の XML 名を設定します。

    - rootName: これはルート要素の名前です。デフォルトは yaml です。
    - elementName: これはシーケンス要素の名前です。デフォルトは要素です。

3. XML 要素名で許可されていない文字をキー名で使用する場合があります。リーダーは、この問題に複数の解決策を提供します。空白、不正な文字、数字で始まるキー名の数字を検索および置換できます。これを使用して、1 つのキー名を完全に別のものに置換することもできます。次

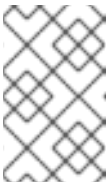
のサンプルコードは、これを行う方法を示しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
            <json:key from="some&key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

- keyWhitspaceReplacement: これは、JSON マップキーの空白の置換文字です。デフォルトでは、これは定義されていないため、リーダーは空白を自動的に検索しません。
- keyPrefixOnNumeric: これは、JSON ノード名が数字で始まる場合に追加する接頭辞文字です。デフォルトでは、これは定義されていないため、リーダーは数字で始まる要素名を検索しません。
- illegalElementNameCharReplacement: JSON 要素名で不正な文字が検出された場合、それらはこの値に置換されます。

4. 次のオプション設定を行うこともできます。

5.
- nullValueReplacement: これは、JSON null 値の置換文字列です。デフォルトは空の文字列です。
- encoding: これは、リーダーによって処理される JSON メッセージ InputStream のデフォルトのエンコードです。デフォルトのエンコードは UTF-8 です。



注記

この機能は非推奨です。代わりに、 **java.io.Reader** を **Smooks.filterSource()** メソッドに指定して、JSON ストリームソースの文字エンコードを管理する必要があります。

6. JSON 設定を読み取るように、Smooks をプログラムで設定するには、**JSONReaderConfigurator** クラスを使用します。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new JSONReaderConfigurator()
        .setRootName("root")
        .setArrayElementName("e"));

// Use Smooks as normal...
```

バグの報告

4.31. JSON データの処理時、XML で許可されていない文字を使用する

XML 要素名で許可されていないキー名の文字を使用するには、リーダーを使用して、空白、不正な文字、および数字で始まるキー名の数字を検索および置換します。これを使用して、1 つのキー名を完全に別のものに置換することもできます。次のサンプルコードは、これを行う方法を示しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
            <json:key from="some&key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

- keyWhitspaceReplacement: これは、JSON マップキーの空白の置換文字です。デフォルトでは、これは定義されていないため、リーダーは空白を自動的に検索しません。
- keyPrefixOnNumeric: これは、JSON ノード名が数字で始まる場合に追加する接頭辞文字です。デフォルトでは、これは定義されていないため、リーダーは数字で始まる要素名を検索しません。
- illegalElementNameCharReplacement: JSON 要素名で不正な文字が検出された場合、それらはこの値に置換されます。

これらの設定はオプションです。

- nullValueReplacement: これは、JSON null 値の置換文字列です。デフォルトは空の文字列です。
- encoding: これは、リーダーによって処理される JSON メッセージ InputStream のデフォルトのエンコードです。デフォルトのエンコードは UTF-8 です。

注記

この機能は非推奨です。代わりに、 **java.io.Reader** を **Smooks.filterSource()** メソッドに指定して、JSON ストリームソースの文字エンコードを管理する必要があります。

バグの報告

4.32. YAML ストリームの設定

手順4.1 タスク

1. 次のように、YAML ファイルを処理するように、リーダーを設定します。

```
<?xml version="1.0"?>
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

    <yaml:reader/>

</smooks-resource-list>
```

2. 複数のドキュメントを含むように、YAML ストリームを設定します。リーダーは、ドキュメント要素をルート要素の子として追加して、これを処理します。1 つの空の YAML ドキュメントを含む XML シリアル化された YAML ストリームは次のようになります。

```
<yaml>
    <document>
    </document>
</yaml>
```

3. **YamlReaderConfigurator** クラスを利用して、YAML 設定を読み取るように、Smooks をプログラムで設定します。

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new YamlReaderConfigurator()
        .setRootName("root")
        .setDocumentName("doc")
        .setArrayElementName("e"))
        .setAliasStrategy(AliasStrategy.REFER_RESOLVE)
        .setAnchorAttributeName("anchor")
        .setAliasAttributeName("alias");

// Use Smooks as normal...
```

バグの報告

4.33. サポート対象の結果タイプ

Smooks は、標準の **JDK StreamResult** および **DOMResult** の結果タイプだけでなく、次の特殊なタイプでも動作します。

- **JavaResult**: この結果タイプを使用して、Smooks Java Bean コンテキストの内容を取得します。
- **ValidationResult**: この結果タイプを使用して、出力を取得します。
- シンプルな結果タイプ: テストを書く際、これを使用します。これは、**StringWriter** をラップする **StreamResult** 拡張です。

バグの報告

4.34. YAML データの処理時、XML で許可されていない文字を使用する

- XML 要素名で許可されていない文字をキー名で使用できます。リーダーは、この問題に複数の解決策を提供します。空白、不正な文字、数字で始まるキー名の数字を検索および置換できます。次のように、1つのキー名を完全に別のものに置換するように、設定できます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

    <yaml:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <yaml:keyMap>
            <yaml:key from="some key">someKey</yaml:key>
            <yaml:key from="some&key" to="someAndKey" />
        </yaml:keyMap>
    </yaml:reader>

</smooks-resource-list>
```

バグの報告

4.35. YAML で XML を置換するためのオプション

- keyWhitspaceReplacement: これは、YAML マップキーの空白の置換文字です。デフォルトでは、これは定義されていません。
- keyPrefixOnNumeric: YAML ノード名が数字で始まる場合は、この接頭辞を追加します。デフォルトでは、これは定義されていません。
- illegalElementNameCharReplacement: YAML 要素名で不正な文字が検出された場合、それらは、この値に置換されます。デフォルトでは、これは定義されていません。

バグの報告

4.36. YAML のアンカーとエイリアス

YAML リーダーは、3 つの異なる方法で *アンカー* と *エイリアス* を処理できます。aliasStrategy 設定オプションを使用して、選択した戦略を定義します。このオプションには、次のいずれかの値を指定できます。

- REFER: リーダーは、アンカーまたはエイリアスを持つ要素に参照属性を作成します。アンカーを持つ要素は、属性値としてアンカーからの名前を含む id 属性を取得します。エイリアスを持つ要素は、属性値としてアンカーの名前も含む ref 属性を取得します。anchorAttributeName および aliasAttributeName プロパティーを設定すると、アンカーおよびエイリアス属性名を定義できます。
- RESOLVE: リーダーは、エイリアスが検出されると、アンカーの値またはデータ構造を解決します。つまり、アンカーの SAX イベントはエイリアス要素の子イベントとして繰り返されます。YAML ドキュメントに多くのアンカーまたはアンカーと実質的なデータ構造が含まれている場合は、メモリーの問題が発生する可能性があります。

- REFER_RESOLVE: これは、REFER と RESOLVE の組み合わせです。アンカーおよびエイリアス属性が設定されますが、アンカーの値またはデータ構造も解決されます。このオプションはアンカーの名前にビジネス上の意味がある場合に役立ちます。

デフォルトでは、YAML リーダーはREFER 戦略を使用します。

バグの報告

4.37. JAVA オブジェクトグラフの変換

1. Smooks は、1 つの *Java オブジェクトグラフ* を別のグラフに変換できます。これを行うには、SAX 処理モデルを使用します。つまり、中間オブジェクトモデルは構成されません。代わりに、ソース Java オブジェクトグラフは、SAX イベントのストリームに直接変換され、ターゲット Java オブジェクトグラフを生成するために使用されます。

   HTML Smooks Report Generator ツールを使用すると、ソースオブジェクトモデルによって生成されるイベントストリームが次のようになることがわかります。

```
<example.srcmodel.Order>
   <header>
      <customerNumber>
         </customerNumber>
        <customerName>
     </customerName>;
   </header>
   <orderItems>
      <example.srcmodel.OrderItem>
         <productId>
        </productId>
         <quantity>
         >/quantity>
         <price>
         </price>
      </example.srcmodel.OrderItem>
   </orderItems>
</example.srcmodel.Order>
```

2. このイベントストリームで Smooks Java Bean リソースを対象とします。この変換を実行するための Smooks 設定 (*smooks-config.xml*) は次のとおりです。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

   <jb:bean BeanId="lineOrder" class="example.trgmodel.LineOrder"
createOnElement="example.srcmodel.Order">
      <jb:wiring property="lineItems" BeanIdRef="lineItems" />
      <jb:value property="customerId" data="header/customerNumber" />
      <jb:value property="customerName" data="header/customerName" />
   </jb:bean<

   <jb:bean BeanId="lineItems" class="example.trgmodel.LineItem[]"
createOnElement="orderItems">
      <jb:wiring BeanIdRef="lineItem" />
```

```
    </jb:bean>


    <jb:bean BeanId="lineItem" class="example.trgmodel.LineItem"
createOnElement="example.srcmodel.OrderItem">
        <jb:value property="productCode" data="example.srcmodel.OrderItem/productId" />
        <jb:value property="unitQuantity" data="example.srcmodel.OrderItem/quantity" />
        <jb:value property="unitPrice" data="example.srcmodel.OrderItem/price" />
    </jb:bean>

</smooks-resource-list>
```

3. ソースオブジェクトモデルは、**org.milyn.delivery.JavaSource** オブジェクトによって Smooks に提供されます。コンストラクターにソースモデルのルートオブジェクトを渡して、このオブジェクトを作成します。結果としての Java ソースオブジェクトは、 **Smooks#filter** メソッドで使用されます。結果のコードは次のとおりです。

```
protected LineOrder runSmooksTransform(Order srcOrder) throws IOException,
SAXException {
    Smooks smooks = new Smooks("smooks-config.xml");
    ExecutionContext executionContext = smooks.createExecutionContext();

    // Transform the source Order to the target LineOrder via a
    // JavaSource and JavaResult instance...
    JavaSource source = new JavaSource(srcOrder);
    JavaResult result = new JavaResult();

    // Configure the execution context to generate a report...
    executionContext.setEventListener(new HtmlReportGenerator("target/report/report.html"));

    smooks.filterSource(executionContext, source, result);

    return (LineOrder) result.getBean("lineOrder");
}
```

バグの報告

4.38. 入力データの文字列操作

CSV および固定長リーダーを使用すると、データが SAX イベントに変換される前に、入力データに文字列操作関数を実行できます。次の機能が利用可能です。

- upper_case: 文字列の大文字バージョンを返します。
- lower_case: 文字列の小文字バージョンを返します。
- cap_first: 最初の単語を大文字にした文字列を返します。
- uncap_first: 最初の単語を大文字にしていない文字列を返します。これは cap_first の反対です。
- capitalize: すべての単語を大文字にした文字列を返します。
- trim: 先頭と末尾の空白を除いた文字列を返します。

- left_trim: 先頭の空白を除いた文字列を返します。
- right_trim: 末尾の空白を除いた文字列を返します。

ポイントセパレーターによって関数を連鎖できます。例は次のとおりです。**trim.upper_case**

フィールドごとに関数を定義する方法は、使用しているリーダーによって異なります。

バグの報告

第5章 検証

5.1. SMOOKS のルール

Smooks では、*ルール* は、一般的な概念であり、特定のカートリッジに固有のものではありません。

他のコンポーネントから *RuleProvider* を設定および参照できます。

注記

ルール機能を使用する唯一のカートリッジは検証カートリッジです。

ルールは、*ruleBase* で一元的に定義されます。1 つの Smooks 設定で多くの ruleBase 定義を参照できます。rulesBase 設定には、名前、ルール **src**、およびルールプロバイダーがあります。

ルールソースの形式は、プロバイダーの実装に完全に依存します。唯一の要件は、個別のルールを参照できるように (1 つのソースのコンテキスト内で)、一意の名前を付けることです。

バグの報告

5.2. SMOOKS でルールを設定する

ruleBase の設定例は次のとおりです。

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="regexAddressing" src="/org/milyn/validation/address.properties"
provider="org.milyn.rules.regex.RegexProvider" />
        <rules:ruleBase name="order" src="/org/milyn/validation/order/rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

バグの報告

5.3. RULES:RULEBASE 設定要素の必須設定

- name: これは、このルールを参照するために、他のコンポーネントによって使用されます。
- src: これは、RuleProvider にとって意味のあるファイルまたはその他のものにすることができます。
- provider: これは実際のプロバイダーの実装です。上記の設定では、正規表現を使用する RuleProvider は 1 つですが、複数の ruleBase 要素を指定して、必要な数の RuleProvider を持つことができます。

バグの報告

5.4. ルールプロバイダー

ルールプロバイダーは **org.milyn.rules.RuleProvider** インターフェイスを実装します。

Smooks は、2 つの RuleProvider 実装をサポートするように、事前設定されています。

- RegexProvider
- MVELProvider

独自の RuleProvider 実装を作成することもできます。

バグの報告

5.5. REGEXPROVIDER

RegexProvider は正規表現を利用します。フィルタリングされるメッセージで選択されたデータフィールドの形式に固有の低レベルルールを定義できます。たとえば、正しい電子メールアドレスが使用されていることを確認するために、特定のフィールドに適用して、構文を検証できます。

バグの報告

5.6. REGEX RULEBASE の設定

1. このコード例を使用して、Regex ruleBase を設定します。

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="customer"
src="/org/milyn/validation/order/rules/customer.properties"
provider="org.milyn.rules.regex.RegexProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

2. 標準の **.properties** ファイル形式で Regex 表現を定義します。次の **customer.properties** Regex ルール定義ファイルの例は、その方法を示しています。

```
# Customer data rules...
customerId=[A-Z][0-9]{5}
customerName=[A-Z][a-z]*, [A-Z][a-z]
```

バグの報告

5.7. MVEL プロバイダー

MVEL プロバイダーを使用すると、ルールを MVEL 式として定義できます。これらの式は、Smooks Javabean コンテキストのコンテンツに実行されます。Smooks Bean コンテキスト内の Java オブジェクトにフィルタリングされているメッセージからバインドデータを取得する必要があります。

MVEL を使用すると、メッセージフラグメントにより複雑なルールを定義できます。(「対象の注文品目フラグメントの製品は注文ヘッダーの詳細で指定された顧客の年齢制限の範囲内ですか?」など)

バグの報告

5.8. MVEL RULEBASE の設定

1. MVEL ruleBase を設定するには、以下のコードを参照してください。

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="order" src="/org/milyn/validation/order/rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

2. MVEL ルールを CSV ファイルに保存する必要があります。これらのファイルを編集する最も簡単な方法は、**LibreOffice Calc** や **Gnumeric** などのスプレッドシートアプリケーションを使用することです。各ルールレコードには、ルール名と MVEL 式が含まれています。

3. コメント行とヘッダー行を作成する場合は、最初のフィールドの前にハッシュ (#) 文字を付けます。

バグの報告

5.9. SMOOKS 検証カートリッジ

Smooks 検証カートリッジは、ルールカートリッジと連携して、*ルールベースのフラグメント検証* を提供します。

これにより、メッセージフラグメントに詳細な検証を実行できます。Smooks のすべてと同様、検証機能はサポートされているすべてのデータ形式で利用できます。つまり、XML データだけでなく、EDI、JSON、CSV などにも強力な検証を行うことができます。

検証設定は、http://www.milyn.org/xsd/smooks/validation-1.0.xsd 設定 namespace で定義されます。

Smooks は多くの異なる *ルールプロバイダータイプ*をサポートしており、これらは、すべて検証カートリッジで使用できます。これらは、それぞれ異なるレベルの検証を提供しますが、すべてまったく同じ方法で設定されます。Smooks Validation Cartridge は、ルールプロバイダーを抽象リソースと見なし、メッセージフラグメントを検証するために対象とすることができます。

バグの報告

5.10. 検証ルールの設定

検証ルールを設定するには、次を指定する必要があります。

- executeOn: これは、ルールが実行されるフラグメントです。
- executeOnNS: これは、**executeOn** が属するフラグメントの namespace です。
- name: これは、適用されるルールの名前です。これは、ルールベースとルール名の組み合わせをドット区切り形式 (つまり、**ruleBaseName.ruleName**) で参照する複合ルール名です。
- onFail: これは、失敗した一致の重大度を決定します。

検証ルールの設定例は次のとおりです。

```
    <validation:rule executeOn="order/header/email" name="regexAddressing.email" onFail="ERROR"
/>
```

バグの報告

5.11. 検証例外の設定

1. Smooks フィルター操作ごとの検証失敗の最大数を設定できます。(この最大値を超えると、例外が出力されます。) OnFail.FATAL で設定された検証は、常に例外を出力し、処理を停止します。

   検証失敗の最大数を設定するには、次のコードを Smooks 設定に追加します。

   ```
   <params>
      <param name="validation.maxFails">5</param>
   </params>
   ```

2. 検証設定の onFail 属性は、実行するアクションを指定します。これは、検証の失敗を報告する方法を決定します。これを利用するには、必要に応じて、次のオプションを変更します。

   - OK: これを使用して、検証を OK として保存します。**ValidationResults.getOks** を呼び出すと、すべての検証警告が返されます。このオプションはコンテンツベースのルーティングに役立ちます。
   - WARN: これを使用して、検証を警告として保存します。**ValidationResults.getWarnings** を呼び出すと、すべての検証警告が返されます。
   - ERROR: これを使用して、検証をエラーとして保存します。**ValidationResults.getErrors** を呼び出すと、すべての検証エラーが返されます。
   - FATAL: これを使用して、検証の失敗が発生するとすぐに、ValidationException を出力します。**ValidationResults.getFatal** を呼び出すと、致命的な検証の失敗が表示されます。

バグの報告

5.12. ルールベース

- ルールベースには、次の形式の複合ルール名を使用します。

```
<ruleProviderName>.<ruleName>
```

  - ruleProviderName は、ルールプロバイダーを識別し、**ruleBase** 要素の **name** 属性にマッピングします。
  - ruleName は、ルールプロバイダーが認識している特定のルールを識別します。これは、**src** ファイルに定義されたルールである可能性があります。

バグの報告

5.13. SMOOKS.FILTERSOURCE

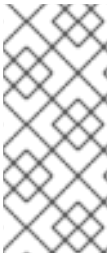
Smooks.filterSource は検証結果を取得します。*filterSource* メソッドが戻ると、ValidationResult インスタンスにすべての検証データが含まれます。

このコードは、Smooks にメッセージフラグメントの検証を実行させる方法を示しています。

```
ValidationResult validationResult = new ValidationResult();

smooks.filterSource(new StreamSource(messageInStream), new
StreamResult(messageOutStream), validationResult);

List<OnFailResult> errors = validationResult.getErrors();
List<OnFailResult> warnings = validationResult.getWarnings();
```

ご覧のとおり、個別の警告とエラーの検証結果は、**ValidationResult** オブジェクトから **OnFailResult** インスタンスの形式で取得できます。各インスタンスは、個別の失敗に関する詳細を提供します。

検証カートリッジを使用すると、検証の失敗に関連するローカライズされたメッセージを指定することもできます。これらのメッセージは、標準の Java ResourceBundle ファイル (**.properties** 形式を使用) で定義できます。



注記

検証メッセージバンドルのベース名は、ルールソースファイル拡張子を削除し、**i18n** という名前のフォルダーを追加すると、ルールソース ("src") から派生します。たとえば、**/org/milyn/validation/order/rules/order-rules.csv** の MVEL ruleBase ソースがある場合、対応する検証メッセージバンドルのベース名は、**/org/milyn/validation/order/rules/i18n/order-rules** になります。

バグの報告

5.14. 検証カートリッジとメッセージ

検証カートリッジを使用すると、ローカライズされたメッセージに **FreeMarker** テンプレートを適用して、メッセージに Bean コンテキストからのコンテキストデータ、および実際のルールの失敗に関するデータを含めることができます。**FreeMarker** ベースのメッセージの前に、**ftl:** を付け、標準の

FreeMarker 表記を使用して、コンテキストデータを参照する必要があります。Bean コンテキストからの Bean は直接参照できますが、**ruleResult** および **path** Bean によって RuleEvalResult およびルール失敗パスを参照できます。

Smooks を使用して、メッセージフラグメントデータの検証を実行する方法を示す RegexProvider ルールを使用する例は次のとおりです。

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.  Customer number must match pattern '${ruleResult.pattern}'.
```

バグの報告

5.15. 検証の種類

Smooks は、2 種類の検証ルールを使用して、2 種類の検証を実行します。

- **.properties** ファイル RuleBase に定義された正規表現を使用したメッセージフィールド値/形式の検証。これは、フィールドが有効な電子メールアドレスであることを検証する場合などに使用できます。
- **.csv** ファイル RuleBase に定義された MVEL 式を使用したビジネスルールの検証。これは、たとえば、注文の注文品目の合計価格 (価格 * 数量) が事前定義されたビジネスルールに違反していないことを検証できます。

バグの報告

5.16. 検証ルールの実行

- 検証ルールを実行するには、例のルートフォルダーに移動し、次を実行します。
  1. **mvn clean install**
  2. **mvn exec:java**

バグの報告

5.17. RULEBASE の例

この例では、注文品目のコレクションを含む XML メッセージがあります。(この機能は、Smooks がサポートする他のすべてのデータ形式でも同様に機能します。):

```
<Order>
 <header>
      <orderId>A188127</orderId>
      <username>user1</username>
      <name>
         <firstname>Bob</firstname>
         <lastname>Bobington</lastname>
      </name>
      <email>bb@awesomemail.com</email>
```

```
    <state>Queensland</state>
  </header>
  <order-item>
    <quantity>1</quantity>
    <productId>364b</productId>
    <title>A Great Movie</title>
    <price>29.95</price>
  </order-item>
  <order-item>
    <quantity>2</quantity>
    <productId>299</productId>
    <title>A Terrible Movie</title>
    <price>29.95</price>
  </order-item>
</Order>
```

バグの報告

5.18. メッセージデータの検証

1. 注文メッセージを処理する際は、多くの検証を行う必要があります。まず、指定されたユーザー名が大文字の後に 5 桁の数字が続く形式 (たとえば、**S12345** または **G54321**) に従っていることを確認します。この検証を実行するには、正規表現を使用する必要があります。

2. 次に、提供された電子メールアドレスが有効な形式であることを確認します。正規表現を使用して、確認します。

3. 各注文品目の productId フィールドが正確に 3 桁の形式 (**123** など) に従っていることを確認します。これを行うには、正規表現を使用します。

4. 最後に、注文品目ごとの合計が 50.00 を超えていないことを確認する必要があります (価格 * 数量は 50.00 を超えていません)。MVEL 式を使用して、この検証を実行します。

バグの報告

5.19. MVEL 式の使用

1. ルールセットで MVEL 式を使用するには、Regex ルールを分割し、2 つの個別の **.properties** ファイルに配置します。

2. これらのルールを **rules** サンプルディレクトリーにドロップします。

3. MVEL 式を **rules** ディレクトリー内の **.csv** ファイルに配置します。

   customer.properties ファイルに含まれる顧客関連の Regex ルールは次のようになります。

```
# Customer data rules...
customerId=[A-Z][0-9]{5}

# Email address...
email=^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$
```

4. 製品関連の Regex ルールを **product.properties** ファイルに挿入します。

```
# Product data rules...
productId=[0-9]{3}
```

5. **order-rules.csv** ファイルに、注文品目の合計チェックを実行するための MVEL 式を挿入します。

注記

.csv ファイルを編集する最も簡単な方法は、**LibeOffice Calc** や **Gnumeric** などのスプレッドシートアプリケーションを使用することです。

6. ルールソースファイルごとにリソースバンドルの **.properties** ファイルを作成します。

注記

これらのファイルの名前は、対応するルールファイルの名前から派生しています。

rules/customer.properties に定義されたルールのメッセージバンドルは、**rules/i18n/customer.properties** ファイルにあります。

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.
<!--  Customer number must begin with an uppercase character, followed by 5 digits. -->
email=ftl:Invalid email address '${ruleResult.text}' at '${path}'.
<!--  Email addresses match pattern '${ruleResult.pattern}'. -->
```

rules/product.properties に定義されたルールのメッセージバンドルは、**rules/i18n/product.properties** ファイルにあります。

```
# Product data rule messages...
productId=ftl:Invalid product ID '${ruleResult.text}' at '${path}'.
<!--  Product ID must match pattern '${ruleResult.pattern}'. -->
```

rules/order-rules.csv に定義されたルールのメッセージバンドルは、**rules/i18n/order-rules.properties** ファイルにあります。

```
# <!--  Order item rule messages. The "orderDetails" and "orderItem" beans are populated by
Smooks bindings - see config in following section. -->
order_item_total=ftl:Order ${orderDetails.orderId}
<!-- contains an order item for product ${orderItem.productId} with a quantity of
${orderItem.quantity} and a unit price of ${orderItem.price}. This exceeds the permitted per
order item total. -->
```

7. この検証をルールに適用します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd"
                      xmlns:validation="http://www.milyn.org/xsd/smooks/validation-1.0.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">
```

```
    <params>
        <!-- Generate a ValidationException if we get more than 5 validation failures... -->
        <param name="validation.maxFails">5</param>
    </params>

    <!-- Define the ruleBases that are used by the validation rules... -->
    <rules:ruleBases>
        <!-- Field value rules using regex... -->
        <rules:ruleBase name="customer" src="rules/customer.properties"
provider="org.milyn.rules.regex.RegexProvider"/>
        <rules:ruleBase name="product" src="rules/product.properties"
provider="org.milyn.rules.regex.RegexProvider"/>

        <!-- Order business rules using MVEL expressions... -->
        <rules:ruleBase name="order" src="rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

    <!-- Capture some data into the bean context - required by the business rule validations... -
->
    <jb:bean beanId="orderDetails" class="java.util.HashMap" createOnElement="header">
        <jb:value data="header/*"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.HashMap" createOnElement="order-item">
        <jb:value data="order-item/*"/>
    </jb:bean>

    <!-- Target validation rules... -->
    <validation:rule executeOn="header/username" name="customer.customerId"
onFail="ERROR"/>
    <validation:rule executeOn="email" name="customer.email" onFail="WARN"/>
    <validation:rule executeOn="order-item/productId" name="product.productId"
onFail="ERROR"/>

    <validation:rule executeOn="order-item" name="order.order_item_total"
onFail="ERROR"/>

</smooks-resource-list>
```

8. 次のコードを使用して、例の **Main** クラスから実行します。

```
protected static ValidationResult runSmooks(final String messageIn) throws IOException,
SAXException, SmooksException {
    // Instantiate Smooks with the config...
    final Smooks smooks = new Smooks("smooks-config.xml");

    try {
        // Create an exec context - no profiles....
        final ExecutionContext executionContext = smooks.createExecutionContext();
        final ValidationResult validationResult = new ValidationResult();

        // Configure the execution context to generate a report...
        executionContext.setEventListener(new
HtmlReportGenerator("target/report/report.html"));
```

```
            // Filter the input message...
            smooks.filterSource(executionContext, new StringSource(messageIn), validationResult);

            return validationResult;
        }
        finally {
            smooks.close();
        }
    }
```

バグの報告

第6章 出力データの生成

6.1. SMOOKS JAVABEAN カートリッジ

Smooks Javabean Cartridge を使用して、メッセージデータから Java オブジェクトを作成および設定できます。XML、EDI、CSV などの Java バインドフレームワークとして純粋に使用できます。ただし、Smooks の Java バインド機能は、他の多くの機能の基礎でもあります。これは、**Smooks** が作成する (およびデータをバインドする) Java オブジェクトを **BeanContext** クラスによって使用できるようにするためです。このクラスは基本的に、Smooks **ExecutionContext** によって、Smooks ビジター実装で使用できる *Java Bean コンテキスト* です。

バグの報告

6.2. JAVABEAN カートリッジの特徴

- テンプレート: これには通常、BeanContext 内のオブジェクトにテンプレートを適用することが含まれます。
- 検証: 通常、ビジネスルールの検証では、BeanContext 内のオブジェクトにルールを適用します。
- メッセージの分割とルーティング: これは、オブジェクト自体を使用して、それらをルーティングするか、テンプレートをそれらに適用して、その操作の結果を新しいファイルにルーティングすることにより、BeanContext 内のオブジェクトから分割メッセージを生成することによって機能します。
- 保持: これらの機能は、データベースにコミットされる Java オブジェクト (エンティティーなど) を作成および投入するための Java バインド機能に依存します。データベースから読み取られたデータは、通常、BeanContext にバインドされます。
- メッセージのエンリッチ: エンリッチデータ (データベースなどから読み取られる) は通常 BeanContext にバインドされ、そこから Java バインド機能自体を含む Smooks の他のすべての機能で使用できます (*式ベースのバインド* に使用できるようにします)。 これにより、Smooks によって生成されたメッセージをエンリッチできます。

バグの報告

6.3. JAVABEAN カートリッジの例

次の例は、この XML に基づいています。

```
<order>
   <header>
      <date>Wed Nov 15 13:45:28 EST 2006</date>
      <customer number="123123">Joe</customer>
   </header>
   <order-items>
      <order-item>
         <product>111</product>
         <quantity>2</quantity>
         <price>8.90</price>
```

```
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

Javabean カートリッジは、http://www.milyn.org/xsd/smooks/javabean-1.4.xsd 設定 namespace で使用されます。(後者のオートコンプリート機能を利用するには、IDE にスキーマをインストールしてください。)

設定例は次のとおりです。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean BeanId="order" class="example.model.Order" createOnElement="#document" />

</smooks-resource-list>
```

この設定は、**example.model.Order** クラスのインスタンスを作成し、**order** という BeanId の下の Bean コンテキストにバインドします。インスタンスは **#document** 要素のメッセージの最初 (つまり、root order 要素の最初) に作成されます。

上記の設定は、**example.model.Order** Bean インスタンスを作成し、Bean コンテキストにバインドします。

バグの報告

6.4. JAVABEAN 要素

- beanId: これは Bean の識別子です。
- class: これは Bean の **完全修飾クラス名** です。
- createOnElement: この属性を使用して、Bean インスタンスをいつ作成するかを制御します。バインド設定 (**jb:bean** 要素の子要素) を使用して、Bean プロパティーの設定を制御できます。
- createOnElementNS: この属性によって createOnElement の namespace を指定できます。

バグの報告

6.5. JAVABEAN 条件

Javabean カートリッジは、次の条件を Java Bean に設定します。

- 引数なしのパブリックコンストラクターがあります。
- *パブリックプロパティーセッターメソッド* があります。これらは、特定の名前形式に従う必要はありませんが、標準のプロパティーセッターメソッド名の形式に従う方がよいでしょう。

- Java Bean プロパティーを直接設定することはできません。

バグの報告

6.6. JAVABEAN カートリッジのデータバインド

これらは、Javabean カートリッジが許可する 3 つのタイプのデータバインドです。

- jb:value: これを使用して、ソースメッセージイベントストリームからターゲット Bean にデータ値をバインドします。
- jb:wiring: これを使用して、別の Bean インスタンスを Bean コンテキストからターゲット Bean の Bean プロパティーに接続します。この設定を使用して、*オブジェクトグラフ* を作成できます (Java オブジェクトインスタンスの緩やかなコレクションとは対照的)。beanId、Java クラスタイプ、またはアノテーションに基づいて、Bean をプラグインできます。
- jb:expression: この設定を使用して、(MVEL 言語で) 式から計算された値をバインドします。簡単な例として、注文品目の合計値を OrderItem Bean にバインドする機能があります (価格 * 数量を計算する式の結果に基づく)。execOnElement 属性式を使用して、式が評価され、結果が結合される要素を定義します。(定義しない場合、式は、親 jb:bean createOnElement の値に基づいて、実行されます。) 対象の要素の値は、**_VALUE** という名前の文字列変数として式で使用できます (アンダースコアに注意してください)。

バグの報告

6.7. データのバインド

1. Order XML メッセージを使用して、完全な XML to Java バインド設定を確認します。その XML メッセージから入力する必要がある Java オブジェクトは次のとおりです (ゲッターとセッターは示されていません)。

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Date date;
    private Long customerNumber;
    private String customerName;
    private double total;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

2. この設定を使用して、注文 XML からオブジェクトモデルにデータをバインドします。

```
<?xml version="1.0"?>
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

(1)   <jb:bean beanId="order" class="com.acme.Order" createOnElement="order">
(1.a)     <jb:wiring property="header" beanIdRef="header" />
(1.b)     <jb:wiring property="orderItems" beanIdRef="orderItems" />
      </jb:bean>

(2)   <jb:bean beanId="header" class="com.acme.Header" createOnElement="order">
(2.a)     <jb:value property="date" decoder="Date" data="header/date">
            <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
          </jb:value>
(2.b)     <jb:value property="customerNumber" data="header/customer/@number" />
(2.c)     <jb:value property="customerName" data="header/customer" />
(2.d)     <jb:expression property="total" execOnElement="order-item" >
            += (orderItem.price * orderItem.quantity);
          </jb:expression>
      </jb:bean>

(3)   <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="order">
(3.a)     <jb:wiring beanType="com.acme.OrderItem" /> <!-- Could also wire using
beanIdRef="orderItem" -->
      </jb:bean>

(4)   <jb:bean beanId="orderItem" class="com.acme.OrderItem" createOnElement="order-
item">
(4.a)     <jb:value property="productId" data="order-item/product" />
(4.b)     <jb:value property="quantity" data="order-item/quantity" />
(4.c)     <jb:value property="price" data="order-item/price" />
      </jb:bean>

</smooks-resource-list>
```

バグの報告

6.8. データ設定のバインド

設定 (1) は、com.acme.Order Bean インスタンス (トップレベルの Bean) の作成規則を定義します。詳細については、次の設定を参照してください。

- メッセージの最初 (order 要素上) で各 Bean インスタンス ((1)、(2)、(3) であり、(4) ではない) を作成する必要があります。これを行うのは、データが取り込まれたモデルにこれらの Bean のインスタンスが 1 つしか存在しないためです。

- 設定 (1.a) と (1.b) は、Header および ListOrderItem Bean インスタンス ((2) と (3)) を注文 Bean インスタンスに接続するための接続設定を定義します (beanIdRef 属性値、および (2) と (3) で定義された beanId 値の参照方法を参照)。(1.a) と (1.b) のプロパティー属性は接続が実行される注文 Bean プロパティーを定義します。

  Bean は、Java クラスタイプ (beanType) に基づいて、または特定のアノテーション (beanAnnotation) でアノテーションを付けて、オブジェクトに接続することもできます。

  設定 (2) は、com.acme.Header Bean インスタンスを作成します。

- 設定 (2.a) は、Header.date プロパティーへの値バインドを定義します。data 属性は、バインド値がソースメッセージから選択される場所を定義することに注意してください。この場合は、ヘッダー/日付要素から来ています。また、decodeParam サブ要素を定義する方法に注意してください。これにより、DateDecoder が設定されます。

- 設定 (2.b) は、値バインド設定を Header.customerNumber プロパティーに定義します。ソースメッセージの要素属性からバインド値を選択するように、データ属性を設定する方法に注意してください。

  設定 (2.b) は、注文合計が計算され、Header.total プロパティーに設定される式バインドも定義します。execOnElement 属性は、この式を注文品目要素で評価 (およびバインド/再バインド) する必要があることを Smooks に指示します。そのため、ソースメッセージに複数の注文品目要素がある場合、この式は注文品目ごとに実行され、新しい合計値が Header.total プロパティーに再バインドされます。式が現在の orderItem 合計を現在の注文合計 (Header.total) に追加する方法に注意してください。

- 設定 (2.d) は、現在の合計に注文品目ごとの合計 (数量 * 価格) を加算することによって、実行中の合計が計算される式バインドを定義します。設定 (3) は、OrderItem インスタンスを保持するための ListOrderItem Bean インスタンスを作成します。

- 設定 (3.a) は、タイプ com.acme.OrderItem (つまり、(4)) のすべての Bean をリストに接続します。この接続がプロパティー属性を定義していないことに注意してください。これは、コレクションに接続しているためです (配列に接続する場合も同様です)。beanType 属性の代わりに beanIdRef 属性を使用して、この接続を実行することもできます。

- 設定 (4) は、OrderItem Bean インスタンスを作成します。createOnElement が注文品目要素に設定されていることに注意してください。これにより、この Bean の新しいインスタンスを注文品目要素ごとに作成できます (また、ListOrderItem (3.a) に接続できます)。

この設定の createOnElement 属性が注文品目要素に設定されていない場合 (たとえば、注文、ヘッダー、または注文品目要素のいずれかに設定されている場合)、1 つの OrderItem Bean インスタンスのみが作成され、バインド設定 ((4.a) など) は、ソースメッセージ内のすべての注文品目要素の Bean インスタンスプロパティーバインドを上書きします。つまり、ソースメッセージに検出された最後の注文品目からの注文品目データを含む OrderItem インスタンスが 1 つだけの ListOrderItem が残ります。

バグの報告

6.9. バインドのヒント

バインドのヒントを次に示します。

- モデル内にインスタンスが 1 つだけ存在する Bean インスタンスのルート要素 (または **#document**) に jb:bean createOnElement を設定します。

  コレクション Bean インスタンス の繰り返し要素に設定します。

警告

この場合は、正しい要素を指定しないと、データが失われる可能性があります。

- jb:value デコーダー: ほとんどの場合、Smooks は jb:value バインドに使用されるデータ型デコーダーを自動的に検出します。ただし、一部のデコーダーでは、設定が必要です (一例として、DateDecoder [**decoder="Date"**])。このような場合は、バインドでデコーダー属性 (およびそのデコーダーのデコードパラメーターを指定するための jb:decodeParam 子要素) を定義する必要があります。
- コレクションにバインドする場合、jb:wiring プロパティーは必要ありません。
- 必要なコレクションタイプを設定するには、jb:bean クラスを定義し、コレクションエントリーに接続します。配列の場合は、jb:bean クラス属性値を角かっこで *後置* するだけです (例: **class=&"com.acme.OrderItem[]"**)。

バグの報告

6.10. DATADECODER/DATAENCODER 実装

DataEncoder は、オブジェクト値を文字列にエンコードおよびフォーマットするためのメソッドを実装します。これらの DataDecoder/DataEncoder 実装が利用可能です。

- Date: 文字列を java.util.Date インスタンスにデコード/エンコードします。
- Calendar: 文字列を java.util.Calendar インスタンスにデコード/エンコードします。
- SqlDate: 文字列を java.sql.Date インスタンスにデコード/エンコードします。
- SqlTime: 文字列を java.sql.Time インスタンスにデコード/エンコードします。
- SqlTimestamp: 文字列を java.sql.Timestamp インスタンスにデコード/エンコードします。

これらの実装は、すべて同じ方法で設定します。

バグの報告

6.11. DATADECODER/DATAENCODER 日付の例

日付の例は次のとおりです。

```
<jb:value property="date" decoder="Date" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</jb:decodeParam>
    <jb:decodeParam name="locale">sv_SE</jb:decodeParam>
</jb:value>
```

バグの報告

6.12. DATADECODER/DATAENCODER SQLTIMESTAMP の例

SqlTimestamp の例は次のとおりです。

```
<jb:value property="date" decoder="SqlTimestamp" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</jb:decodeParam>
```

```
    <jb:decodeParam name="locale">sv</jb:decodeParam>
</jb:value>
```

バグの報告

6.13. DATADECODER/DATAENCODER DECODEPARAM の例

decodeParam 形式は、日付形式の ISO 8601 標準に基づいています。ロケールの decodeParam 値はアンダースコアで区切られた文字列であり、1 つ目のトークンはロケールの ISO 言語コードであり、2 つ目のトークンは ISO 国コードです。この decodeParam は、言語と国の 2 つの個別のパラメーターとして指定することもできます。

```
<jb:value property="date" decoder="Date" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</jb:decodeParam>
    <jb:decodeParam name="locale-language">sv</jb:decodeParam>
    <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

バグの報告

6.14. 数値ベースの DATADECODER/DATAENCODER 実装

複数の数値ベースの DataDecoder/DataEncoder 実装が利用可能です。

- BigDecimalDecoder: これを使用して、文字列を java.math にデコード/エンコードします。BigDecimal インスタンス。
- BigIntegerDecoder: これを使用して、文字列を java.math にデコード/エンコードします。BigInteger インスタンス。
- DoubleDecoder: これを使用して、文字列を java.lang.Double インスタンス (プリミティブを含む) にデコード/エンコードします。
- FloatDecoder: これを使用して、文字列を java.lang.Float インスタンス (プリミティブを含む) にデコード/エンコードします。
- IntegerDecoder: これを使用して、文字列を java.lang.Integer インスタンス (プリミティブを含む) にデコード/エンコードします。
- LongDecoder: これを使用して、文字列を java.lang.Long インスタンス (プリミティブを含む) にデコード/エンコードします。
- ShortDecoder: これを使用して、文字列を java.lang.Short インスタンス (プリミティブを含む) にデコード/エンコードします。

これらの実装は、すべて同じ方法で設定します。

バグの報告

6.15. 数値ベースの DATADECODER/DATAENCODER の例

BigDecimal の例は次のとおりです。

```
<jb:value property="price" decoder="BigDecimal" data="orderItem/price">
    <jb:decodeParam name="format">#,###.##</jb:decodeParam>
    <jb:decodeParam name="locale">en_IE</jb:decodeParam>
</jb:value>
```

バグの報告

6.16. 数値ベースの DATADECODER/DATAENCODER 整数の例

```
<jb:value property="percentage" decoder="Integer" data="vote/percentage">
    <jb:decodeParam name="format">#%</jb:decodeParam>
</jb:value>
```

バグの報告

6.17. 数値ベースの DATADECODER/DATAENCODER DECODEPARAM の例

decodeParam 形式は、NumberFormat パターンの構文に基づいています。ロケールの decodeParam 値はアンダースコアで区切られた文字列であり、1 つ目のトークンはロケールの ISO 言語コードであり、2 つ目のトークンは ISO 国コードです。この decodeParam は、言語と国の 2 つの個別のパラメーターとして指定することもできます。例を参照してください。

```
<jb:value property="price" decoder="Double" data="orderItem/price">
    <jb:decodeParam name="format">#,###.##</jb:decodeParam>
    <jb:decodeParam name="locale-language">sv</jb:decodeParam>
    <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

バグの報告

6.18. マッピングデコーダーをバインドに使用する

1. 次のように、入力メッセージのデータに基づいて、異なる値をオブジェクトモデルにバインドするように、マッピングデコーダーを設定します。

```
<jb:value property="name" decoder="Mapping" data="history/@warehouse">
    <jb:decodeParam name="1">Dublin</jb:decodeParam>
    <jb:decodeParam name="2">Belfast</jb:decodeParam>
    <jb:decodeParam name="3">Cork</jb:decodeParam>
</jb:value>
```

2. 入力データ値 1 は、name プロパティに値 Dublin としてマッピングされます。値 2 と 3 についても同様です。

バグの報告

6.19. 列挙型デコーダー

*列挙型デコーダー*は、マッピングデコーダーの特殊なバージョンです。通常、データ入力値が列挙型の値/名前に正確にマッピングされている場合、列挙型は (特定の設定を必要とせずに) 自動的にデコードされます。ただし、そうでない場合は、入力データ値から列挙型の値/名前へのマッピングを定義する必要があります。

バグの報告

6.20. 列挙型デコーダーの例

1. 次の例では、入力メッセージのヘッダー/優先度フィールドに、**LOW**、**MEDIUM**、および **HIGH** の値が含まれています。これらを、それぞれ **NOT_IMPORTANT**、**IMPORTANT**、および **VERY_IMPORTANT** の LineOrderPriority 列挙型の値にマッピングする必要があります。

```
<jb:value property="priority" data="header/priority" decoder="Enum">
    <jb:decodeParam
name="enumType">example.trgmodel.LineOrderPriority</jb:decodeParam>
    <jb:decodeParam name="LOW">NOT_IMPORTANT</jb:decodeParam>
    <jb:decodeParam name="MEDIUM">IMPORTANT</jb:decodeParam>
    <jb:decodeParam name="HIGH">VERY_IMPORTANT</jb:decodeParam>
</jb:value>
```

2. マッピングが必要な場合は、enumType decodeParam を使用して、列挙型を指定します。

バグの報告

6.21. BEANCONTEXT 設定

デフォルトでは、Bean を作成したフラグメント (createOnElement) が処理された後、最初の Bean を除く Smooks 設定内のすべての Bean が BeanContext から削除されます。(つまり、Bean は createOnElement フラグメントの start/visitBefore で BeanContext に追加され、end/visitAfter で BeanContext から削除されます。)

デフォルトでは、このルールは、Smooks 設定に設定された最初の Bean を除くすべての Bean に適用されます。(最初の Bean は、BeanContext に保持される唯一の Bean であるため、メッセージが処理された後、アクセスできます。)

この動作を変更するには、jb:bean 要素の retain configuration 属性を使用します。この属性を使用すると、Smooks BeanContext 内で Bean の保持を手動で制御できます。

バグの報告

6.22. JAVABEAN カートリッジのアクション

Java Bean カートリッジ:

- ソース/入力メッセージストリームから文字列値を抽出します。

- **decoder** および **decodeParam** 設定に基づいて、文字列値をデコードします (これらが定義されていない場合は、デコーダーを反射的に解決しようとします)。
- ターゲット Bean にデコードされた値を設定します。

バグの報告

6.23. 文字列データの前処理

デコードする前に、文字列データ値を *前処理* する必要がある場合があります。この例としては、数値 876592.00 が "876_592!00"jb:bean 要素として表されるなど、数値デコードのロケール設定でサポートされていない文字がソースデータに含まれている場合があります。この値を (たとえば) double 値としてデコードするには、アンダースコアを削除し、感嘆符をピリオドに置換します。valuePreprocess decodeParam を指定できます。これは、String 値をデコードする前に適用できる単純な式です。

バグの報告

6.24. 前処理の例

この例では、数値のデコードの問題に対する解決策を示します。

```
<!-- A bean property binding example: -->
<jb:bean beanId="orderItem" class="org.milyn.javabean.OrderItem" createOnElement="price">
    <jb:value property="price" data="price" decoder="Double">
        <jb:decodeParam name="valuePreprocess">value.replace("_", "").replace("!", ".")
</jb:decodeParam>
    </jb:value>
</jb:bean>
```

```
<!-- A direct value binding example: -->
<jb:value beanId="price" data="price" decoder="BigDecimal">
    <jb:decodeParam name="valuePreprocess">value.replace("_", "").replace("!", ".")
</jb:decodeParam>
</jb:value>
```

文字列データ値は、値変数名によって式で参照されます。(式は、値文字列を操作して、文字列を返す任意の有効な MVEL 式にすることができます。)

バグの報告

6.25. JAVABEAN カートリッジとファクトリー

Javabean カートリッジを使用すると、 *ファクトリー* を使用して、Bean を作成できます。このような場合、*パラメーターなしのパブリックコンストラクター* を使用する必要はありません。class 属性で実際のクラス名を定義する必要はありません。オブジェクトのインターフェイスのいずれでも十分です。ただし、そのインターフェイスのメソッドのみにバインドできます。(ファクトリーを定義する場合も、Bean 定義で class 属性を必ず設定する必要があります。)

ファクトリー定義は、bean 要素の factory 属性に設定されます。デフォルトのファクトリー定義言語は次のようになります。

```
some.package.FactoryClass#staticMethod{.instanceMethod}
```

この基本的な定義言語を使用して、Bean を作成するために、Smooks が呼び出すパラメーターなしの静的パブリックメソッドを定義します。(instanceMethod 部分はオプションです。設定すると、静的メソッドから返されるオブジェクトで呼び出されるメソッドが定義され、Bean が作成されます。{ } 文字は、その部分がオプションであることを示すためだけにあり、実際の定義から除外する必要があります。)

バグの報告

6.26. 静的ファクトリーメソッドを使用して、ARRAYLIST オブジェクトをインスタンス化する

1. 次の例に従って、静的ファクトリーメソッドを使用して、ArrayList オブジェクトをインスタンス化します。

```
<jb:bean
  beanId="orders"
  class="java.util.List"
  factory="some.package.ListFactory#newList"
  createOnElement="orders"
>
    <!-- ... bindings -->
</jb:bean>
```

some.package.ListFactory#newList ファクトリー定義は、Bean を作成するために、**some.package.ListFactory** クラスで **newList** メソッドを呼び出す必要があります。クラス属性は、Bean を List オブジェクトとして定義します。List オブジェクトの特定の種類 (ArrayList か LinkedList か) は、ListFactory 自体によって決定されます。

2. 次の追加の例を確認してください。

```
<jb:bean
  beanId="orders"
  class="java.util.List"
  factory="some.package.ListFactory#getInstance.newList"
  createOnElement="orders"
>
    <!-- ... bindings -->
</jb:bean>
```

これは、静的メソッド **getInstance** を使用して、ListFactory のインスタンスを取得する必要があること、および ListFactory オブジェクト上でメソッド **newList** を呼び出して、List オブジェクトを作成する必要があることを定義しています。この構成では、*シングルトンファクトリー* を使用できます。

バグの報告

6.27. 定義言語の宣言

使用する定義言語を宣言するには、次の方法があります。

- 各定義言語には、エイリアスを設定できます。たとえば、MVEL には、エイリアス mvel があります。特定のファクトリー定義に MVEL を使用することを定義するには、定義の前に mvel: を置きます。たとえば、mvel:some.package.ListFactory.getInstance().newList() です。デフォルトの基本言語のエイリアスは basic です。

- 言語をグローバルデフォルトとして設定するには、factory.definition.parser.class グローバルパラメーターを、使用する言語の FactoryDefinitionParser インターフェイスを実装するクラスの完全なクラスパスに設定する必要があります。**:** を含むデフォルト言語による定義がある場合は、例外を回避するために、その定義の前に **default:** を付ける必要があります。

- 使用する言語の FactoryDefinitionParser インターフェイスを実装するクラスの完全なクラスパスを設定できます。(例: org.milyn.javabean.factory.MVELFactoryDefinitionParser:some.package.ListFactory.getInstance() 通常、これはテスト目的のみに使用してください。言語のエイリアスを定義する方がはるかに優れています。

バグの報告

6.28. 独自の定義言語の使用

1. 独自の言語を定義するには、org.milyn.javabean.factory.FactoryDefinitionParser インターフェイスを実装します。org.milyn.javabean.factory.MVELFactoryDefinitionParser または org.milyn.javabean.factory.BasicFactoryDefinitionParser の例を確認してください。

2. 定義言語のエイリアスを定義するには、エイリアス名を持つ org.milyn.javabean.factory.Alias アノテーションを FactoryDefinitionParser クラスに追加します。

3. Smooks がエイリアスを見つけるには、クラスパスのルートに META-INF/smooks-javabean-factory-definition-parsers.inf ファイルを作成する必要があります。T

バグの報告

6.29. MVEL 言語

MVEL には、基本的なデフォルトの定義言語と比較して、いくつかの利点があります。これにより、Bean コンテキストのオブジェクトをファクトリーオブジェクトとして使用でき、パラメーターを使用して、ファクトリーメソッドを呼び出すことができます。これらのパラメーターは、定義内に定義することも、Bean コンテキストからのオブジェクトにすることもできます。MVEL を使用するには、エイリアス **mvel** を使用するか、**factory.definition.parser.class** グローバルパラメーターを **org.milyn.javabean.factory.MVELFactoryDefinitionParser** に設定します。

バグの報告

6.30. MVEL の例

```
<smooks-resource-list
  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

   <jb:bean
      beanId="orders"
```

```
        class="java.util.List"
        factory="mvel:some.package.ListFactory.getInstance().newList()"
        createOnElement="orders"
    >
        <!-- ... bindings -->
    </jb:bean>

</smooks-resource-list>
```

バグの報告

6.31. MVEL を使用したリストオブジェクトの抽出

MVEL を使用して、Bean コンテキスト内の既存の Bean から List オブジェクトを抽出するには、以下の例を参照してください。(この例の Order オブジェクトには、注文明細を追加するために使用する必要があるリストを返すメソッドがあります。)

```
<smooks-resource-list
    xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean
        beanId="order"
        class="some.package.Order"
        createOnElement="order"
    >
        <!-- ... bindings -->
    </jb:bean>

    <!--
            The factory attribute uses MVEL to access the order
            object in the bean context and calls its getOrderLines()
            method to get the List. This list is then added to the bean
            context under the beanId 'orderLines'
    -->
    <jb:bean
        BeanId="orderLines"
        class="java.util.List"
        factory="mvel:order.getOrderLines()"
        createOnElement="order"
    >
        <jb:wiring BeanIdRef="orderLine" />
    </jb:bean>

    <jb:bean
        BeanId="orderLine"
        class="java.util.List"
        createOnElement="order-line"
    >
        <!-- ... bindings -->
    </jb:bean>

</smooks-resource-list>
```

注記

配列オブジェクトはサポートされていません。ファクトリーが配列を返す場合、Smooks は例外を出力します。

バグの報告

6.32. JB:VALUE プロパティー

キーと値のペアをマップにバインドする際、バインドの jb:value プロパティー属性が定義されていない場合は、選択したノードの名前がマップエントリーキーとして使用されます (beanClass はマップです)。@ 文字を前に置くと、jb:value プロパティー属性を定義することもできます。残りの値は、選択されたノードのアトリビュート名を定義し、そこからマップキーが選択されます。

バグの報告

6.33. JB:VALUE プロパティーの例

```
<root>
    <property name="key1">value1</property>
    <property name="key2">value2</property>
    <property name="key3">value3</property>
</root>
```

設定上:

```
<jb:bean BeanId="keyValuePairs" class="java.util.HashMap" createOnElement="root">
    <jb:value property="@name" data="root/property" />
</jb:bean>
```

これにより、キーセット [key1, key2, key3] を持つ 3 つのエントリーを持つ HashMap が作成されます。

注記

@ 文字表記は、bean 接続では機能しません。カートリッジは、単に "@" 文字を含むプロパティー属性の値をマップエントリーキーとして使用します。

バグの報告

6.34. 仮想オブジェクトモデル

独自の Bean クラスを記述しなくても、*仮想オブジェクトモデル* を作成できます。この仮想モデルは、マップとリストのみを使用して、作成されます。これは、2 つの処理ステップの間に Javabean カートリッジを使用する場合に便利です。たとえば、モデルドリブン変換の一部として、XML to Java to XML または XML to Java to EDI。

バグの報告

6.35. 仮想オブジェクトモデルの例

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Bind data from the message into a Virtual Object model in the bean context....
    -->
    <jb:bean beanId="order" class="java.util.HashMap" createOnElement="order">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
    </jb:bean>
    <jb:bean beanId="header" class="java.util.HashMap" createOnElement="order">
        <jb:value property="date" decoder="Date" data="header/date">
            <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</jb:decodeParam>
        </jb:value>
        <jb:value property="customerNumber" decoder="Long" data="header/customer/@number" />
        <jb:value property="customerName" data="header/customer" />
        <jb:expression property="total" execOnElement="order-item" >
            header.total + (orderItem.price * orderItem.quantity);
        </jb:expression>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="order">
        <jb:wiring beanIdRef="orderItem" />
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.HashMap" createOnElement="order-item">
        <jb:value property="productId" decoder="Long" data="order-item/product" />
        <jb:value property="quantity" decoder="Integer" data="order-item/quantity" />
        <jb:value property="price" decoder="Double" data="order-item/price" />
    </jb:bean>

    <!--
    Use a FreeMarker template to perform the model driven transformation on the Virtual Object
Model...
    -->
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/orderA-to-orderB.ftl</ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

上記で、仮想モデル (マップ) のデコーダー属性が常にどのように定義されるかに注意してください。これは、Smooks にはマップへのデータバインドのデコードタイプを自動検出する方法がないためです。仮想モデルにバインドされた型指定された値が必要な場合は、適切なデコーダーを指定する必要があります。この場合、デコーダーが指定されていないと、Smooks は単純にデータを文字列として仮想モデルにバインドします。

バグの報告

6.36. 複数のデータエンティティーを 1 つのバインドにマージする

式ベースのバインド (**jb:expression**) を使用すると、複数のデータエンティティーを 1 つのバインドにマージできます。Javabean カートリッジは、Smooks DataDecoder を使用して、選択したデータ要素/属性からオブジェクトを作成します。次に、Bean コンテキストに直接追加します。

バグの報告

6.37. 値バインド

ValueBinder クラスは、値バインドを実行するビジターです。値バインド XML 設定は、http://www.milyn.org/xsd/smooks/javabean-1.4.xsd 上の Smooks 1.3 以降の JavaBean スキーマの一部です。値バインドの要素は値です。

バグの報告

6.38. 値バインドの属性

ValueBinder クラスには、次の属性があります。

- beanId: 作成されたオブジェクトが Bean コンテキストでバインドされる ID。
- data: バインドするデータ値のデータセレクター。(例: **order/orderid** または **order/header/@date**
- dataNS: データセレクターの namespace
- デコーダー: 値を String から別の型に変換するための DataDecoder 名。DataDecoder は、decodeParam 要素で設定できます。
- default: 選択したデータが null または空の文字列の場合のデフォルト値。

バグの報告

6.39. 値バインドの例

注文メッセージを例として使用します。注文番号、名前、日付を値オブジェクトとして整数と文字列の形式で取得するように、設定されます。

メッセージ入力:

```
<order xmlns="http://x">
    <header>
        <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
        <customer number="123123">Joe</customer>
        <privatePerson></privatePerson>
    </header>
    <order-items>
        <!-- .... -->
    </order-items>
 </order>
```

設定:

```
<?xml version="1.0"?>
 <smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:value
       beanId="customerName"
       data="customer"
       default="unknown"
    />

    <jb:value
       beanId="customerNumber"
       data="customer/@number"
       decoder="Integer"
    />

   <jb:value
       beanId="orderDate"
       data="date"
       dateNS="http://y"
       decoder="Date"
    >
          <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z yyyy</jb:decodeParam>
          <jb:decodeParam name="locale-language">en</jb:decodeParam>
          <jb:decodeParam name="locale-country">IE</jb:decodeParam>
   </jb:value>

 </smooks-resource-list>
```

バグの報告

6.40. プログラムによる値バインドの例

値バインダーは、org.milyn.javabean.Value オブジェクトを使用して、プログラムで設定できます。

```
//Create Smooks. normally done globally!
Smooks smooks = new Smooks();

//Create the Value visitors
Value customerNumberValue = new Value( "customerNumber",
"customer/@number").setDecoder("Integer");
Value customerNameValue = new Value( "customerName", "customer").setDefault("Unknown");

//Add the Value visitors
smooks.addVisitors(customerNumberValue);
smooks.addVisitors(customerNameValue);

//And the execution code:
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Integer customerNumber = (Integer) result.getBean("customerNumber");
String customerName = (String) result.getBean("customerName");
```

バグの報告

6.41. SMOOKS での JAVA バインド

Java バインド設定は、Bean 設定クラスを使用して、Smooks にプログラムで追加できます。このクラスを使用すると、特定のクラスで Java バインドを実行するために、Smooks インスタンスをプログラムで設定できます。グラフを作成するには、Bean を Bean にバインドして、Bean インスタンスのグラフを作成するだけです。Bean クラスは、Fluent API (すべてのメソッドが Bean インスタンスを返す) を使用するため、設定を簡単に結合して、Bean 設定のグラフを作成できます。

バグの報告

6.42. JAVA バインドの例

この注文メッセージの例は、対応する Java オブジェクトモデルにバインドする方法を示しています。

メッセージ入力:

```
<order xmlns="http://x">
    <header>
        <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
        <customer number="123123">Joe</customer>
        <privatePerson></privatePerson>
    </header>
    <order-items>
        <order-item>
            <product>111</product>
            <quantity>2</quantity>
            <price>8.90</price>
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

Java モデル (getter/setter を含まない):

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Long customerNumber;
    private String customerName;
}

public class OrderItem {
    private long productId;
```

```
    private Integer quantity;
    private double price;
}
```

設定コード:

```
Smooks smooks = new Smooks();

Bean orderBean = new Bean(Order.class, "order", "/order");

orderBean.bindTo("header",
    orderBean.newBean(Header.class, "/order")
        .bindTo("customerNumber", "header/customer/@number")
        .bindTo("customerName", "header/customer")
    ).bindTo("orderItems",
    orderBean.newBean(ArrayList.class, "/order")
        .bindTo(orderBean.newBean(OrderItem.class, "order-item")
            .bindTo("productId", "order-item/product")
            .bindTo("quantity", "order-item/quantity")
            .bindTo("price", "order-item/price"))
    );

smooks.addVisitors(orderBean);
```

実行コード:

```
 JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Order order = (Order) result.getBean("order");
```

匿名の Factory クラスを定義および使用する例は次のとおりです。

```
Bean orderBean = new Bean(Order.class, "order", "/order", new Factory<Order>() {

 public Order create(ExecutionContext executionContext) {
  return new Order();
 }

});
```

バグの報告

6.43. ORG.MILYN.JAVABEAN.GEN.CONFIGGENERATOR ユーティリティークラス

Javabean カートリッジには、バインド設定テンプレートの生成に使用できる org.milyn.javabean.gen.ConfigGenerator ユーティリティークラスが含まれています。このテンプレートは、バインドを定義するための基礎として使用できます。

バグの報告

6.44. ORG.MILYN.JAVABEAN.GEN.CONFIGGENERATOR の例

コマンドラインから:

```
$JAVA_HOME/bin/java -classpath <classpath> org.milyn.javabean.gen.ConfigGenerator -c
<rootBeanClass> -o <outputFilePath> [-p <propertiesFilePath>]
```

- **-c** コマンドライン引数は、バインド設定が生成されるモデルのルートクラスを指定します。
- **-o** コマンドライン引数は、生成された設定出力のパスとファイル名を指定します。
- **-p** コマンドライン引数は、追加のバインドパラメーターを指定するオプションのバインド設定ファイルのパスとファイル名を指定します。
- オプションの **-p** プロパティーファイルパラメーターを使用すると、追加の設定パラメーターを指定できます。
- packages.included: セミコロンで区切られたパッケージのリスト。これらのパッケージに一致するクラス内のすべてのフィールドは、生成されたバインド設定に含まれます。
- packages.excluded: セミコロンで区切られたパッケージのリスト。これらのパッケージに一致するクラス内のすべてのフィールドは、生成されたバインド設定から除外されます。

バグの報告

6.45. バインド設定のプログラミング

ターゲットクラスに org.milyn.javabean.gen.ConfigGenerator ユーティリティークラスを実行した後は、これらのタスクを実行して、ソースデータモデルにバインド設定が機能するようにする必要があります。

1. jb:bean 要素ごとに、Bean インスタンスの作成に使用するイベント要素に createOnElement 属性を設定します。
2. jb:value データ属性を更新して、その BFean プロパティーのバインドデータを提供するイベント要素/属性を選択します。
3. jb:value デコーダーの属性を確認してください。実際のプロパティータイプによっては、すべてが設定されるわけではありません。これらは手動で設定する必要があります。日付フィールドなど、一部のバインドでデコーダーの jb:decodeParam サブ要素を設定する必要がある場合があります。
4. バインド設定要素 (jb:value および jb:wiring) を再確認し、生成された設定ですべての Java プロパティーがカバーされていることを確認します。

バグの報告

6.46. 変換の設定

1. セレクターの値を決定する際、HTML レポートツールにアクセスします。これは、Smooks から見た (セレクターが適用される) 入力メッセージモデルを視覚化するために役立ちます。

2. ソースデータを使用して、レポートを生成しますが、変換設定は空です。レポートでは、設定を追加する必要があるモデルを確認できます。設定を 1 つずつ追加し、レポートを再実行して、設定が適用されていることを確認します。

3. 設定を 1 つずつ追加し、レポートを再実行して、設定が適用されていることを確認します。

4. その結果、次のような設定が生成されます (**$TODO$** トークンに注意してください)。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean beanId="order" class="org.milyn.javabean.Order" createOnElement="$TODO$">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray" beanIdRef="orderItemsArray" />
    </jb:bean>

    <jb:bean beanId="header" class="org.milyn.javabean.Header"
createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long" data="$TODO$" />
        <jb:value property="customerName" decoder="String" data="$TODO$" />
        <jb:value property="privatePerson" decoder="Boolean" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean beanId="orderItems_entry" class="org.milyn.javabean.OrderItem"
createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray" class="org.milyn.javabean.OrderItem[]"
createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItemsArray_entry" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray_entry" class="org.milyn.javabean.OrderItem"
createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

</smooks-resource-list>
```

注記

Smooks.filterSource メソッドを呼び出した後の JavaResult インスタンスの正確な内容に関する保証はありません。このメソッドを呼び出した後、JavaResult インスタンスには、任意のビジター実装によって追加できる Bean コンテキストの最終的な内容が含まれます。

バグの報告

6.47. JB:RESULT の設定例

Smooks 設定で *jb:result* 設定を使用すると、JavaResult で返される Bean セットを制限できます。次の設定例では、ResultSet に **order** Bean のみを保持するように、Smooks に指示します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <!-- Capture some data from the message into the bean context... -->
    <jb:bean beanId="order" class="com.acme.Order" createOnElement="order">
        <jb:value property="orderId" data="order/@id"/>
        <jb:value property="customerNumber" data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItems" beanIdRef="orderItems"/>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="order">
        <jb:wiring beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="com.acme.OrderItem" createOnElement="order-item">
        <jb:value property="itemId" data="order-item/@id"/>
        <jb:value property="productId" data="order-item/product"/>
        <jb:value property="quantity" data="order-item/quantity"/>
        <jb:value property="price" data="order-item/price"/>
    </jb:bean>

    <!-- Only retain the "order" bean in the root of any final JavaResult. -->
    <jb:result retainBeans="order"/>

</smooks-resource-list>
```

この設定を適用した後、非 **order** Bean の結果に対する JavaResult.getBean (String) メソッドの呼び出しは、null を返します。これは、他の Bean インスタンスが **order** グラフに接続しているため、上記の例のような場合に機能します。

注記

Smooks v1.2 の時点で、JavaSource インスタンスが Smooks.filterSource メソッドに (フィルターソースインスタンスとして) 提供される場合、Smooks は、JavaSource を使用して、その Smooks.filterSource 呼び出しの ExecutionContect に関連付けられた Bean コンテキストを構成します。これは、JavaSource Bean インスタンスの一部が JavaResult に表示される可能性があることを意味します。

バグの報告

第7章 テンプレート

7.1. SMOOKS テンプレート

Smooks で利用できるテンプレートは、FreeMarker (http://freemarker.org) と XSL (http://www.w3.org/Style/XSL/) の 2 種類です。

これらのテクノロジーは、Smooks フィルタリングプロセスのコンテキスト内で使用できます。つまり、次のとおりです。

- *フラグメント* ごとにソースメッセージに適用できます。これは、メッセージ全体に適用されるフラグメントベースの変換プロセスとは対照的です。フラグメントごとに適用すると、SOAP メッセージにヘッダーを追加する場合など、非常に特定のポイントでメッセージにデータを挿入する必要がある場合に役立ちます。この場合、プロセスは、残りの部分を中断せずに、関心のあるフラグメントを対象とすることができます。

- Javabean Cartridge などの他の Smooks テクノロジーを利用できます。これは、Bean コンテキストへのメッセージデータのデコードおよびバインドに使用できます。次に、**FreeMarker** テンプレート内から、そのデコードされたデータを参照できます。(Smooks はデータを **FreeMarker** で使用できるようにします。)

- 巨大なメッセージストリーム (数ギガバイトのサイズ) を処理するために使用できます。同時に、単純な処理モデルと小さなメモリーフットプリントを維持します。

- *分割メッセージフラグメント* を生成するために使用できます。これらは、Enterprise Service Bus 上の物理的または論理的なエンドポイントにルーティングできます。

バグの報告

7.2. FREEMARKER テンプレート

FreeMarker は非常に強力な *テンプレートエンジン* です。Smooks は、テキストベースのコンテンツを生成する手段として **FreeMarker** を使用できます。その後、このコンテンツをメッセージストリームに挿入できます (このプロセスは *フラグメントベースの変換* と呼ばれます)。このプロセスを使用して、メッセージフラグメントを分割して、その後、別のプロセスにルーティングすることもできます。

設定 namespace http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd を使用して、Smooks の **FreeMarker** テンプレートを設定します。次に、XSD を使い始めるために、統合開発環境で設定します。

バグの報告

7.3. FREEMARKER テンプレートの例

- インラインテンプレートの例:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template><!--<orderId>${order.id}</orderId>--></ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

- FreeMarker 外部テンプレートの例:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

- Smooks が結果の出力にさまざまな操作を実行できるように、<ftl:freemarker> 設定に <use> 要素 を追加します。以下の例を参照してください。

```
<ftl:freemarker applyOnElement="order">
    <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    <ftl:use>
        <ftl:inline directive="insertbefore" />
    </ftl:use>
</ftl:freemarker>
```

バグの報告

7.4. SMOOKS でのインライン

*インライン*は、その名前が示すように、テンプレート結果を **Smooks.filterSource** の結果にインラインできます。多くのディレクティブがサポートされています。

- addto: これにより、テンプレート結果が対象の要素に追加されます。
- replace (デフォルト): テンプレート結果を使用して、対象の要素を置換します。これは、<use> 要素が設定されていない場合の <ftl:freemarker> 設定のデフォルトの動作です。
- insertbefore: これにより、テンプレート結果が対象の要素の前に追加されます。
- insertafter: これにより、テンプレート結果が対象の要素の後に追加されます。

バグの報告

7.5. FTL:BINDTO 要素

ftl:bindTo 要素を使用すると、テンプレート結果を Smooks Bean コンテキストにバインドできます。結果は、ルーティングに使用されるコンポーネントなど、他の Smooks コンポーネントからアクセスできます。特に、巨大なメッセージを小さなものに分割する場合に役立ちます。分割フラグメントは、別のプロセスにルーティングできます。

バグの報告

7.6. FTL:BINDTO の例

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <jms:router routeOnElement="order-item" beanId="orderItem_xml" destination="queue.orderItems"
/>

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Bind the templating result into the bean context, from where
                 it can be accessed by the JMSRouter (configured above). -->
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

バグの報告

7.7. FTL:OUTPUTTO 要素

ftl:outputTo を使用して、出力結果を直接 **OutputStreamResource** クラスに書き込むことができます。これは、巨大なメッセージを小さなメッセージに分割するためのもう 1 つの役立つ機能です。

バグの報告

7.8. FTL:OUTPUTTO の例

テンプレートの結果を **OutputStreamSource** に書き込む例:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.3.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!-- Create/open a file output stream. This is written to by the freemarker template (below).. -->
    <file:outputStream openOnElement="order-item" resourceName="orderItemSplitStream">
        <file:fileNamePattern>order-${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>
        <file:destinationDirectoryPattern>target/orders</file:destinationDirectoryPattern>
        <file:listFileNamePattern>order-${order.orderId}.lst</file:listFileNamePattern>

        <file:highWaterMark mark="3"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this freemarker template,
    outputting the result to the "orderItemSplitStream" OutputStream, which is the file
    output stream configured above.
    -->
     <ftl:freemarker applyOnElement="order-item">
```

```
        <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Output the templating result to the "orderItemSplitStream" file output stream... -->
            <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

注記

包括的なチュートリアルは、http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples にあります。

バグの報告

7.9. 変換の設定

1. セレクターの値を決定する際、HTML レポートツールにアクセスします。これは、Smooks から見た (セレクターが適用される) 入力メッセージモデルを視覚化するために役立ちます。
2. ソースデータを使用して、レポートを生成しますが、変換設定は空です。レポートでは、設定を追加する必要があるモデルを確認できます。設定を 1 つずつ追加し、レポートを再実行して、設定が適用されていることを確認します。
3. 設定を 1 つずつ追加し、レポートを再実行して、設定が適用されていることを確認します。
4. その結果、次のような設定が生成されます (**$TODO$** トークンに注意してください)。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean beanId="order" class="org.milyn.javabean.Order" createOnElement="$TODO$">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray" beanIdRef="orderItemsArray" />
    </jb:bean>

    <jb:bean beanId="header" class="org.milyn.javabean.Header"
createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long" data="$TODO$" />
        <jb:value property="customerName" decoder="String" data="$TODO$" />
        <jb:value property="privatePerson" decoder="Boolean" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean beanId="orderItems_entry" class="org.milyn.javabean.OrderItem"
createOnElement="$TODO$">
```

```
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray" class="org.milyn.javabean.OrderItem[]"
createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItemsArray_entry" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray_entry" class="org.milyn.javabean.OrderItem"
createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$" />
        <jb:value property="quantity" decoder="Integer" data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

</smooks-resource-list>
```

注記

Smooks.filterSource メソッドを呼び出した後の JavaResult インスタンスの正確な内容に関する保証はありません。このメソッドを呼び出した後、JavaResult インスタンスには、任意のビジター実装によって追加できる Bean コンテキストの最終的な内容が含まれます。

バグの報告

7.10. FREEMARKER と JAVA BEAN カートリッジ

FreeMarker NodeModel は、強力で使いやすいですが、パフォーマンスの点でトレードオフがあります。W3C DOM の構成は低コストではありません。また、必要なデータがすでに抽出され、Java オブジェクトモデルに投入されている場合があります。たとえば、データを一連のオブジェクトとして Java Message Service エンドポイントにルーティングする必要がある場合などです。

NodeModel の使用が実用的でない場合は、Java Bean カートリッジを使用して、適切な Java オブジェクトまたは仮想モデルを設定します。このモデルは、**FreeMarker** テンプレートプロセスで使用できます。

バグの報告

7.11. NODEMODEL の例

次の例は、NodeModel 要素を設定する方法を示しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.3.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
```

```
    <!-- Extract and decode data from the message. Used in the freemarker template (below). -->
    <jb:bean beanId="order" class="java.util.Hashtable" createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long" data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.Hashtable" createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-item/price"/>
    </jb:bean>

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--<orderitem id="${order.orderItem.itemId}" order="${order.orderId}">
    <customer>
        <name>${order.customerName}</name>
        <number>${order.customerNumber?c}</number>
    </customer>
    <details>
        <productId>${order.orderItem.productId}</productId>
        <quantity>${order.orderItem.quantity}</quantity>
        <price>${order.orderItem.price}</price>
    </details>
</orderitem>-->
        </ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

注記

拡張された例は、http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples にあります。

バグの報告

7.12. プログラムによる設定

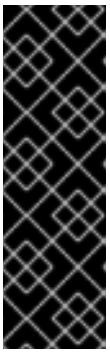
FreeMarker テンプレート設定は、Smooks インスタンスにプログラムで追加できます。これを行うには、次のように、**FreeMarkerTemplateProcessor** インスタンスを追加および設定します。この例では、Java バインドと **FreeMarker** テンプレートの設定を Smooks に追加します。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new Bean(OrderItem.class, "orderItem", "order-item").bindTo("productId", "order-item/product/@id"));
smooks.addVisitor(new FreeMarkerTemplateProcessor(new TemplatingConfiguration("/templates/order-tem.ftl")), "order-item");

// Then use Smooks as normal... filter a Source to a Result etc...
```

バグの報告

7.13. XSL テンプレート

- Smooks で XSL テンプレートを使用するには、統合開発環境で http://www.milyn.org/xsd/smooks/xsl-1.1.xsd XSD を設定してください。



重要

JBoss SOA では、ルートフラグメントに適用される 1 つの XSLT のみが Smooks 設定に含まれている場合、フラグメントフィルターはバイパスされます。XSLT は直接適用されます。これは、パフォーマンス上の理由で行われ、**enableFilterBypass** というパラメーターを追加し、**false** に設定すると、無効にすることができます。

```
<param name="enableFilterBypass">false</param>
```

バグの報告

7.14. XSL の例

次の例は、XSL テンプレートを設定する方法を示しています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd">

    <xsl:xsl applyOnElement="#document">
        <xsl:template><!--<xxxxxx/>--></xsl:template>
    </xsl:xsl>

</smooks-resource-list>
```

FreeMarker と同様、他のタイプの外部テンプレートは、<xsl:template> 要素内の URI 参照を使用して、設定できます。

バグの報告

7.15. XSL サポートの注意点

次の場合を除き、Smooks を使用して、XSL テンプレートを実行する理由はありません。

- メッセージ全体の変換ではなく、フラグメント変換を実行する必要があります。
- 分割、保持など、メッセージに追加の操作を行うには、他の Smooks 機能を使用する必要があります。
- XSL テンプレートは、DOM フィルターのみによってサポートされます。SAX フィルターではサポートされていません。これにより、(適用される XSL によっては) XSL の SAX ベースのアプリケーションと比較して、パフォーマンスが低下する可能性があります。

- Smooks は、メッセージフラグメントごとに XSL テンプレートを適用します。これは XSL のフラグメント化に非常に役立ちますが、スタンドアロンコンテキスト用に記述されたテンプレートが、変更なく、Smooks で自動的に機能するとは想定しないでください。このため、Smooks は、テンプレートを (ルート DOM 要素ではなく) DOM ドキュメントノードに適用するという点で、ドキュメントルートノードを対象とする XSL を異なる方法で処理します。

- ほとんどの XSL には、ルート要素に一致するテンプレートが含まれています。Smooks はフラグメントごとに XSL を適用するため、これはもはや有効ではありません。代わりに、コンテキストノード (つまり、対象のフラグメント) に一致するテンプレートがスタイルシートに含まれていることを確認してください。

バグの報告

7.16. 潜在的な問題: XSLT は、外部では機能するが、SMOOKS 内では機能しない

これは、場合によって発生する可能性があり、通常、その原因は次のシナリオのいずれかです。

- ドキュメントルートノードへの絶対パス参照を使用するテンプレートがスタイルシートに含まれている場合は、Smooks フラグメントベースの処理モデルで問題が発生します。これは、Smooks が誤った要素を対象としているためです。この問題を解決するには、Smooks が対象としているコンテキストノードに一致するテンプレートが XSLT に含まれていることを確認してください。

- *SAX と DOM の処理の比較*: 「いいね」と「いいね」が比較されているわけではありません。現状では、Smooks は XSL を処理するための DOM ベースの処理モデルのみをサポートしています。正確な比較を行うには、Smooks 外で XSL テンプレートを実行する際、*DOMSource* (namespace を認識するもの) を使用します。(特定の XSL プロセッサーが、SAX または DOM を使用して、XSL テンプレートを適用しようとしても、常に同じ出力を生成するとはかぎりません。)

バグの報告

第8章 出力データのエンリッチ

8.1. すぐに使えるエンリッチ方法

出力データをエンリッチするための 3 つの方法が製品に含まれています。

JDBC データソース

JDBC データソースを使用して、データベースにアクセスし、SQL ステートメントを使用して、データベースの読み取りと書き込みを行います。この機能は、Smooks Routing Cartridge によって提供されます。SQL を使用したデータベースへのルーティングに関するセクションを参照してください。

エンティティーの保持

エンティティー保持フレームワーク (Ibatis、Hibernate、または任意の JPA 互換フレームワークなど) を使用して、データベースにアクセスし、そのクエリー言語または CRUD メソッドを使用して、データベースの読み取りまたは書き込みを行います。

DAO

カスタムデータアクセスオブジェクト (DAO) を使用して、データベースにアクセスし、その CRUD メソッドを使用して、データベースの読み取りまたは書き込みを行います。

バグの報告

8.2. ハイバネーションの例

処理するデータは、XML 注文メッセージです。必要に応じて、入力データは、CSV、JSON、EDI、Java、またはその他の構造化/階層データ形式にすることもできます。データ形式に関係なく、同じ原則が適用されます。JPA 準拠のフレームワークでも、同じ原則に従います。

```
<order>
   <ordernumber>1</ordernumber>
   <customer>123456</customer>
   <order-items>
      <order-item>
         <product>11</product>
         <quantity>2</quantity>
      </order-item>
      <order-item>
         <product>22</product>
         <quantity>7</quantity>
      </order-item>
   </order-items>
</order>
```

バグの報告

8.3. HIBERNATE エンティティー

以下は、Hibernate 関数のエンティティーのスナップショットです。

```
@Entity
@Table(name="orders")
public class Order {

    @Id
    private Integer ordernumber;

    @Basic
    private String customerId;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List orderItems = new ArrayList();

    public void addOrderLine(OrderLine orderLine) {
        orderItems.add(orderLine);
    }

    // Getters and Setters....
}

@Entity
@Table(name="orderlines")
public class OrderLine {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name="orderid")
    private Order order;

    @Basic
    private Integer quantity;

    @ManyToOne
    @JoinColumn(name="productid")
    private Product product;

    // Getters and Setters....
}

@Entity
@Table(name = "products")
@NamedQuery(name="product.byId", query="from Product p where p.id = :id")
public class Product {

    @Id
    private Integer id;

    @Basic
    private String name;
```

```
    // Getters and Setters....
}
```

バグの報告

8.4. 注文の処理と保持

1. XML 注文メッセージを処理および保持するには、注文データを Order エンティティー (Order、OrderLine、および Product) にバインドする必要があります。これを行うには、Java Binding フレームワークを使用して、Order エンティティーと OrderLine エンティティーを作成および投入します。

2. 各 OrderLine インスタンスを Order インスタンスに接続します。

3. 各 OrderLine インスタンスで、関連する注文明細の Product エンティティーを検索および接続する必要があります。

4. 最後に、下記のように、Order インスタンスを挿入 (保持) します。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"
                      xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean beanId="order" class="example.entity.Order" createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber" />
        <jb:value property="customerId" data="customer" />
        <jb:wiring setterMethod="addOrderLine" beanIdRef="orderLine" />
    </jb:bean>

    <jb:bean beanId="orderLine" class="example.entity.OrderLine" createOnElement="order-
item">
        <jb:value property="quantity" data="quantity" />
        <jb:wiring property="order" beanIdRef="order" />
        <jb:wiring property="product" beanIdRef="product" />
    </jb:bean>

    <dao:locator beanId="product" lookupOnElement="order-item" onNoResult="EXCEPTION"
uniqueResult="true">
        <dao:query>from Product p where p.id = :id</dao:query>
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer" />
        </dao:params>
    </dao:locator>

    <dao:inserter beanId="order" insertOnElement="order" />

</smooks-resource-list>
```

5. クエリー文字列の代わりに名前付きクエリー **productById** を使用する場合、DAO ロケーターの設定は次のようになります。

```
<dao:locator beanId="product" lookupOnElement="order-item" lookup="product.byId"
```

```
onNoResult="EXCEPTION" uniqueResult="true">
    <dao:params>
        <dao:value name="id" data="product" decoder="Integer"/>
    </dao:params>
</dao:locator>
```

バグの報告

8.5. SESSIONREGISTER オブジェクトによる SMOOKS の実行

次のコードは Smooks を実行します。Smooks 内から Hibernate セッションにアクセスできるように、**SessionRegister** オブジェクトが使用されます。

```
Smooks smooks = new Smooks("smooks-config.xml");

ExecutionContext executionContext = smooks.createExecutionContext();

// The SessionRegister provides the bridge between Hibernate and the
// Persistence Cartridge. We provide it with the Hibernate session.
// The Hibernate Session is set as default Session.
DaoRegister register = new SessionRegister(session);

// This sets the DAO Register in the executionContext for Smooks
// to access it.
PersistenceUtil.setDAORegister(executionContext, register);

Transaction transaction = session.beginTransaction();

smooks.filterSource(executionContext, source);

transaction.commit();
```

バグの報告

8.6. DAO による注文の保持

1. DAO で注文を保持するには、以下のコード例を確認してください。この例では、注文情報を含む XML ファイルを読み取ります (これは、EDI、CSV などでも同様に機能します)。Javabean カートリッジを使用して、XML データを一連のエンティティー Bean にバインドします。製品エンティティーを検索し、注文品目 (製品要素) 内の製品の ID を使用して、注文エンティティー Bean にバインドします。最後に、注文 Bean が保持されます。

   注文 XML メッセージは次のようになります。

```
<order>
    <ordernumber>1</ordernumber>
    <customer>123456</customer>
    <order-items>
        <order-item>
            <product>11</product>
            <quantity>2</quantity>
```

```
        </order-item>
        <order-item>
            <product>22</product>
            <quantity>7</quantity>
        </order-item>
    </order-items>
</order>
```

2. Order エンティティーを保持するには、次の例のようなカスタム DAO を使用します。

```
@Dao
public class OrderDao {

    private final EntityManager em;

    public OrderDao(EntityManager em) {
        this.em = em;
    }

    @Insert
    public void insertOrder(Order order) {
        em.persist(order);
    }
}
```

このクラスを見ると、@Dao および @Insert アノテーションに気付くはずです。@Dao アノテーションは、OrderDao が DAO オブジェクトであることを宣言します。@Insert アノテーションは、insertOrder メソッドを使用して、Order エンティティーを挿入する必要があることを宣言します。

3. 次の例のように、カスタム DAO を使用して、Product エンティティーを検索します。

```
@Dao
public class ProductDao {

    private final EntityManager em;

    public ProductDao(EntityManager em) {
        this.em = em;
    }

    @Lookup(name = "id")
    public Product findProductById(@Param("id")int id) {
        return em.find(Product.class, id);
    }
}
```

このクラスを見ると、@Lookup および @Param アノテーションに気付くはずです。@Lookup アノテーションは、ProductDao#findByProductId メソッドを使用して、Product エンティティーを検索することを宣言します。@Lookup アノテーションの name パラメーターは、そのメソッドのルックアップ名参照を設定します。名前が宣言されていない場合は、メソッド名が使用されます。オプションの @Param アノテーションを使用すると、パラメーターに名前を付けることができます。これにより、Smooks と DAO の間の抽象化が向上します。@Param アノテーションを宣言しない場合、パラメーターはその位置によって解決されます。

4. 上記のように注文を設定すると、結果の Smooks 設定は次のようになります。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"
                      xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean BeanId="order" class="example.entity.Order" createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber"/>
        <jb:value property="customerId" data="customer"/>
        <jb:wiring setterMethod="addOrderLine" BeanIdRef="orderLine"/>
    </jb:bean>

    <jb:bean BeanId="orderLine" class="example.entity.OrderLine" createOnElement="order-
item">
        <jb:value property="quantity" data="quantity"/>
        <jb:wiring property="order" BeanIdRef="order"/>
        <jb:wiring property="product" BeanIdRef="product"/>
    </jb:bean>

    <dao:locator BeanId="product" dao="product" lookup="id" lookupOnElement="order-item"
onNoResult="EXCEPTION">
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer"/>
        </dao:params>
    </dao:locator>

    <dao:inserter BeanId="order" dao="order" insertOnElement="order"/>

</smooks-resource-list>
```

5. 次のコードを使用して、Smooks を実行します。

```
Smooks smooks=new Smooks("./smooks-configs/smooks-dao-config.xml");
ExecutionContext executionContext=smooks.createExecutionContext();

// The register is used to map the DAO's to a DAO name. The DAO name isbe used in
// the configuration.
// The MapRegister is a simple Map like implementation of the DaoRegister.
DaoRegister<object>register = MapRegister.builder()
        .put("product",new ProductDao(em))
        .put("order",new OrderDao(em))
        .build();

PersistenceUtil.setDAORegister(executionContext,mapRegister);

// Transaction management from within Smooks isn't supported yet,
// so we need to do it outside the filter execution
EntityTransaction tx=em.getTransaction();
tx.begin();

smooks.filter(new StreamSource(messageIn),null,executionContext);

tx.commit();
```

バグの報告

第9章 GROOVY スクリプト

9.1. GROOVY

Groovy は、Java 仮想マシン用のアジャイルで動的な言語であり、Java の長所に基づいて、構築されていますが、Python、Ruby、Smalltalk などの言語に触発された追加の強力な機能を備えています。

詳細については、http://groovy.codehaus.org/ を参照してください。

バグの報告

9.2. GROOVY の例

Groovy スクリプトのサポートは、*設定 namespace* (http://www.milyn.org/xsd/smooks/groovy-1.1.xsd) で提供されます。 この namespace は、DOM および SAX ベースのスクリプトをサポートします。以下の例を参照してください。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

    <g:groovy executeOnElement="xxx">
        <g:script>
        <!--
        //Rename the target fragment element from "xxx" to "yyy"...
        DomUtils.renameElement(element, "yyy", true, true);
        -->
        </g:script>
    </g:groovy>

</smooks-resource-list>
```

バグの報告

9.3. GROOVY のヒント

- *ビジットされた要素* は、適切な名前の変数によってスクリプトで使用できます。element(要素名と同じ変数名でも使用できますが、後者の名前が英数字に制限されている場合に限ります。)
- *前に実行/後に実行*: デフォルトでは、visitAfter イベントの発生時、スクリプトが実行されます。executeBefore 属性を **true** に設定して、visitBefore で実行するように、指示します。
- *コメント/CDATA スクリプトラップ*: スクリプトに特殊な XML 文字が含まれている場合は、スクリプトを **XML Comment** または **CDATA** セクションでラップできます。

バグの報告

9.4. IMPORTS

imports 要素を使用して、*インポート* を追加します。多くのクラスが自動的にインポートされます。

- **org.milyn.xml.DomUtils**
- **org.milyn.javabean.context.BeanContext**.Smooks 1.3 以降のみ。
- **org.milyn.javabean.repository.BeanRepository**
- **org.w3c.dom.***
- **groovy.xml.dom.DOMCategory**
- **groovy.xml.dom.DOMUtil**
- **groovy.xml.DOMBuilder**

バグの報告

9.5. GROOVY で DOM と SAX の混合を使用する

Groovy は、DOM と SAX の混合モデルをサポートしています。**Groovy** の DOM ユーティリティーを使用して、対象のメッセージフラグメントを処理できます。SAX フィルターが使用されている場合も、**Groovy** スクリプトは、DOM の要素を受け取ります。これにより、SAX フィルターを使用する **Groovy** スクリプトがはるかに簡単になり、ストリーミング方式で巨大なメッセージを処理する機能が維持されます。

バグの報告

9.6. DOM と SAX の混合のヒント

注意事項:

- デフォルトモードのみで使用できます (つまり、executeBefore が **false** に等しい場合)。executeBefore が **true** であるように設定されている場合、この機能は、利用できません。つまり、**Groovy** スクリプトは、SAXElement のみにアクセスできます。
- writeFragment は、DOM フラグメントを **Smooks.filterSource StreamResult** に書き込むために、呼び出す必要があります。
- この DOM 構成機能を使用すると、パフォーマンスのオーバーヘッドが発生します。(巨大なメッセージを処理することはできますが、少し時間がかかる場合があります。妥協点は使いやすさとパフォーマンスの間です。)

バグの報告

9.7. DOM と SAX の混合の例

手順9.1 タスク

1. 次のサンプルのような XML メッセージを考えてみましょう。

```
<shopping>
    <category type="groceries">
        <item>Chocolate</item>
        <item>Coffee</item>
    </category>
    <category type="supplies">
        <item>Paper</item>
        <item quantity="4">Pens</item>
    </category>
        <category type="present">
        <item when="Aug 10">Kathryn's Birthday</item>
    </category>
</shopping>
```

2. 上記の買い物リストの消耗品カテゴリーを変更するには、さらに 2 つのペンを追加します。これを行うには、単純な **Groovy** スクリプトを記述し、メッセージの <category> 要素を対象とします。

3. その結果、スクリプトは単純にカテゴリー内の <item> 要素を反復し、カテゴリータイプが「消耗品」で品目が「ペン」の場合、数量は 2 つ増加します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
  xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

  <core:filterSettings type="SAX" />

  <g:groovy executeOnElement="category">
    <g:script>
    <!--
      use(DOMCategory) {
       // Modify "supplies": we need an extra 2 pens...
       if (category.'@type' == 'supplies') {
           category.item.each { item ->
               if (item.text() == 'Pens') {
                   item['@quantity'] = item.'@quantity'.toInteger() + 2;
               }
           }
       }
    }

    // When using the SAX filter, we need to explicitly write the fragment
    // to the result stream...
    writeFragment(category);
    -->
    </g:script>
  </g:groovy>

</smooks-resource-list>
```

バグの報告

第10章 出力データのルーティング

10.1. 出力データオプション

Smooks は、メッセージフラグメントの分割とルーティングについて、さまざまなオプションをサポートしています。メッセージをフラグメントに分割し、これらのフラグメントをさまざまなエンドポイント (ファイル、JMS など) にルーティングする機能は、非常に重要な機能です。Smooks は、次の機能とともにこれを提供します。

- **Basic Fragment Splitting**: メッセージにダム分割を実行する必要がある場合があります。つまり、メッセージ内のすべての注文品目フラグメントを分割し、ファイルにルーティングします。ダム分割とは、ルーティング前にメッセージ階層の他の部分からのデータをマージするなど、ルーティング前に、分割メッセージフラグメントにどのような変換も実行する必要がないことを意味します。(たとえば、ルーティングの前に注文品目フラグメントに顧客の詳細情報を追加します。)基本的な分割とルーティングでは、分割するメッセージフラグメントの XPath を定義し、未変更の分割メッセージフラグメントをルーティングするルーティングコンポーネント (JBoss ESB や Camel など) を定義するだけです。

- **複雑なフラグメント分割**: 基本的なフラグメント分割は、多くのユースケースで機能し、ほとんどの分割およびルーティングソリューションが提供するものです。Smooks は、ルーティングが適用される前に、分割フラグメントデータに変換を実行できるようにして、基本的な分割機能を拡張します。たとえば、顧客詳細の注文情報を各注文品目情報とマージしてから、ルーティング注文品目分割フラグメントルーティングを実行します。

- **In Stream Splitting and Routing** (巨大なメッセージのサポート): Smooks はストリーム内でルーティングを実行できるため (完全なメッセージを処理した後、ルーティング用にまとめることはできません)、ギガバイト単位のサイズの巨大なメッセージストリームの処理に対応できます。

- **Multiple Splitting and Routing**: 入力メッセージストリームの 1 つのフィルタリングパスで複数のメッセージフラグメント (さまざまな形式の XML、EDI、Java など) を条件付きで分割し、さまざまなエンドポイントにルーティングします。たとえば、OrderItem Java オブジェクトインスタンスを $1,000 を超える値の注文品目の **HighValueOrdersValidation** JMS キューにルーティングし、すべての注文品目 (無条件) を XML/JSON としてログ用の HTTP エンドポイントにルーティングします。

バグの報告

10.2. APACHE CAMEL を使用したルーティング

1. メッセージフラグメントを Apache Camel エンドポイントにルーティングするには、http://www.milyn.org/xsd/smooks/camel-1.4.xsd 設定 namespace の camel:route 設定を使用します。

2. Smooks 設定で以下を指定して、Camel エンドポイントにルーティングします。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
        xmlns:camel="http://www.milyn.org/xsd/smooks/camel-1.4.xsd">

  <!-- Create some bean instances from the input source... -->
  <jb:bean beanId="orderItem"  ... ">
    <!-- etc... See Smooks Java Binding docs -->
  </jb:bean>
```

```
  <!-- Route bean to camel endpoints... -->
  <camel:route beanId="orderItem">
    <camel:to endpoint="direct:slow" if="orderItem.priority == 'Normal'" />
    <camel:to endpoint="direct:express" if="orderItem.priority == 'High'" />
  </camel:route>

</smooks-resource-list>
```

上記の例では、Javabeans は Smooks BeanContext から Camel エンドポイントにルーティングされます。これらの同じ Bean にテンプレート (FreeMarker など) を適用し、Bean (XML や CSV など) の代わりにテンプレート結果をルーティングすることもできます。

routeOnElement.

バグの報告

第11章 パフォーマンスチューニング

11.1. パフォーマンスチューニングのヒント

Smooks オブジェクトをキャッシュおよび再利用します。

Smooks の初期化には時間がかかるため、再利用することが重要です。

可能な場合は、リーダーインスタンスをプールする

一部のリーダーは作成に非常にコストがかかるため、パフォーマンスが大幅に向上する可能性があります。

可能な場合は、SAX フィルタリングを使用する

SAX 処理は、**DOM** 処理よりもはるかに高速であり、使用するメモリーも少なくなります。大きなメッセージを処理するには、必須です。すべての Smooks カートリッジが **SAX** 互換であることを確認します。

デバッグログをオフにする

Smooks は、コードの一部で集中的なデバッグログを実行します。これにより、処理のオーバーヘッドが大幅に増加し、スループットが低下する可能性があります。

重要

ログをまったく設定しないと、デバッグログステートメントが実行される可能性があることに注意してください。

HTMLReportGenerator は、開発環境のみで使用してください。

HTMLReportGenerator を有効にすると、パフォーマンスのオーバーヘッドが大きくなり、メッセージが大きい場合は、**OutOfMemory** 例外が発生する可能性もあります。

コンテキストセレクター

コンテキストセレクターは明らかにパフォーマンスに悪影響を及ぼす可能性があります。たとえば、**"a/b/c/d/e"** のようなセレクターの一致を評価するには、**"d/e"** のようなセレクターよりも多くの処理が必要になることは明らかです。明らかに、データモデルが深いセレクターを必要とする状況がありますが、そうでない場合は、セレクターのパフォーマンスを最適化するようにしてください。

可能な場合は、仮想 Bean モデルの使用を回避し、マップの代わりに Bean を作成してください。マップへのデータの作成と追加は、*単純な古い Java オブジェクト* (**POJO**) を作成し、セッターメソッドを呼び出すよりもはるかに時間がかかります。

バグの報告

第12章 テスト

12.1. 単体テスト

- Smooks で単体テストを行うには、次の例に従います。

```
public class MyMessageTransformTest
{
    @Test
    public void test_transform() throws IOException, SAXException
    {
        Smooks smooks = new Smooks(
            getClass().getResourceAsStream("smooks-config.xml") );

        try {
            Source source = new StreamSource(
                getClass().getResourceAsStream("input-message.xml" ) );
            StringResult result = new StringResult();

            smooks.filterSource(source, result);

            // compare the expected xml with the transformation result.
            XMLUnit.setIgnoreWhitespace( true );
            XMLAssert.assertXMLEqual(
                new InputStreamReader(
                getClass().getResourceAsStream("expected.xml")),
                new StringReader(result.getResult()));
        } finally {
            smooks.close();
        }
    }
}
```

上記のテストケースでは、**XMLUnit** と呼ばれるソフトウェアを使用します (詳細については、http://xmlunit.sourceforge.net を参照)。

注記

上記のテストには、次の **Maven** 依存関係が必要でした。

```
<dependency>
    <groupId>xmlunit</groupId>
    <artifactId>xmlunit</artifactId>
    <version>1.1</version>
</dependency>
```

バグの報告

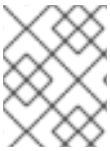
第13章 一般的なユースケース

13.1. 巨大なメッセージの処理のサポート

Smooks は、巨大なメッセージに対して次のタイプの処理をサポートしています。

- **One-to-one transformation**: これは、巨大なメッセージをソース形式 (XML など) からターゲット形式 (EDI、CSV、XML など) に変換するプロセスです。
- **Splitting and routing**: 巨大なメッセージを任意の形式 (EDI、XML、Java など) の小さな (より消費しやすい) メッセージに分割し、それらの小さなメッセージをさまざまな宛先タイプ (ファイル、JMS、データベース) にルーティングします。
- **Persistence**: 巨大なメッセージのコンポーネントをデータベースに保持し、そこからより簡単にクエリーおよび処理できるようにします。Smooks 内では、これを分割とルーティング (データベースへのルーティング) の形式と見なします。

上記のすべては、コードを記述せずに (つまり、宣言的な方法で) 可能です。また、ソースメッセージを 1 回のパスで処理し、分割およびルーティングを並行して実行できます (さらに、異なるタイプおよび異なる形式の複数の宛先へのルーティングも実行します)。



注記

Smooks で巨大なメッセージを処理する場合は、パフォーマンスを向上させるために、SAX フィルターを使用していることを確認してください。

バグの報告

13.2. FREEMARKER を使用して、巨大なメッセージを変換する

巨大なメッセージを別の形式の 1 つのメッセージに変換して処理するには、複数の FreeMarker テンプレートをソースメッセージイベントストリームに適用し、Smooks.filterSource Result ストリームに出力します。これは、次の 2 つの方法のいずれかで実行できます。

- モデルに FreeMarker と NodeModel を使用します。
- モデルに FreeMarker と Java オブジェクトモデルを使用します。Javabean Cartridge を使用して、メッセージ内のデータからモデルを構成できます。

バグの報告

13.3. 巨大なメッセージと NODEMODEL

メッセージが巨大な場合は、複数の NodeModel を識別して、実行時のメモリーフットプリントを可能なかぎり低くする必要があります。完全なメッセージは大きすぎてメモリーに保持できないため、 1 つのモデルを使用して、メッセージを処理することはできません。注文メッセージの場合は、2 つのモデルがあります。1 つはメインの注文データ用であり、もう 1 つは注文品目データ用です。

一度にメモリーに保存されるほとんどのデータは、メインの注文データと注文品目の 1 つです。NodeModel がネストされているため、Smooks は注文データ NodeModel に注文品目 NodeModel からのデータがまったく含まれないようにします。また、Smooks がメッセージをフィルタリングすると、注文品目 NodeModel は、すべての注文品目で上書きされます (つまり、収集されません)。

バグの報告

13.4. 複数の NODEMODEL を取得するように、SMOOKS を設定する

1. FreeMarker テンプレートで使用する複数の NodeModel を取得するように、Smooks を設定するには、*DomModelCreator* ビジターを設定する必要があります。各モデルのルートノードを対象とする必要があります。Smooks は、これを SAX フィルタリング (巨大なメッセージを処理するために重要) でも使用できるようにすることに注意してください。

   これは、メッセージの NodeModel を作成するための Smooks 設定です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM for the "complete" message - there are "mini" DOMs
    for the NodeModels below)....
    -->
    <core:filterSettings type="SAX" defaultSerialization="false" />

    <!--
    Create 2 NodeModels. One high level model for the "order"
    (header etc) and then one for the "order-item" elements...
    -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!-- FreeMarker templating configs to be added below... -->
```

2. 次に、次の FreeMarker テンプレートを適用します。

   - 注文品目までは含まれない、注文 **header** の詳細を出力するテンプレート。
   - salesorder の item 要素を生成するための、注文品目ごとのテンプレート。
   - メッセージを閉じるためのテンプレート。

   Smooks では、2 つの FreeMarker テンプレートを定義すると、これを実装できます。1 つ目は上記のポイント 1 と 3 (結合) をカバーし、2 つ目は項目要素をカバーします。

3. 最初の FreeMarker テンプレートを適用します。注文品目要素を対象としており、次のようになります。

```
<ftl:freemarker applyOnElement="order-items">
        <ftl:template><!--<salesorder>
    <details>
        <orderid>${order.@id}</orderid>
        <customer>
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
```

```
    </details>
    <itemList>
    <?TEMPLATE-SPLIT-PI?>
    </itemList>
</salesorder>-->
        </ftl:template>
    </ftl:freemarker>
```

?TEMPLATE-SPLIT-PI? 処理命令は、テンプレートを分割する場所を Smooks に指示し、テンプレートの最初の部分を注文品目要素の先頭に出力し、残りの部分を注文品目要素の末尾に出力します。item 要素テンプレート (2 つ目のテンプレート) は間に出力されます。

4. 2 つ目の FreeMarker テンプレートを適用します。これは、ソースメッセージ内のすべての注文品目要素の末尾に品目要素を出力します。

```
<ftl:freemarker applyOnElement="order-item">
        <ftl:template><!-- <item>
    <id>${.vars["order-item"].@id}</id>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
</item>-->
        </ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

2 つ目のテンプレートは注文品目要素の末尾で起動するため、最初のテンプレートの **?TEMPLATE-SPLIT-PI?** 処理命令の場所に出力を効果的に生成します。2 つ目のテンプレートは、NodeModel の **order** でデータを参照することもできることに注意してください。

5. 任意のクロージングテンプレートを適用します。

注記

巨大なメッセージの 1 対 1 変換を実行するこのアプローチが機能する理由は、一度にメモリーにあるオブジェクトが注文ヘッダーの詳細と現在の注文品目の詳細 (仮想オブジェクトモデル内) のみであるためです。ソースメッセージ内のすべてのデータへのフルアクセスが常に必要なほど変換があいまいな場合、たとえば、メッセージですべての注文品目の順序を逆にする (または並べ替える) 必要がある場合は、明らかに機能します。ただし、このような場合は、注文の詳細と品目をデータベースにルーティングし、データベースのストレージ、クエリー、およびページング機能を使用して、変換を実行するオプションがあります。

バグの報告

13.5. メッセージ分割の要件

巨大なメッセージは、個別に処理できる小さなメッセージに分割すると、処理できます。たとえば、注文メッセージ内の注文品目を分割し、(コンテンツに基づいて) 処理のために別の部門またはパートナーにルーティングする必要がある場合は、小さなメッセージ (メッセージサイズは関係ない場合があります) で分割とルーティングが必要になることもあります。これらの条件下では、次の例のように、さまざまな宛先で必要なメッセージ形式も異なる場合があります。

- **destination1**: ファイルシステム経由の XML が必要です。

- **destination2**: JMS キューによって Java オブジェクトが必要です。
- **destination3**: データベースなどのテーブルからメッセージを取得します。
- **destination4**: JMS キュー経由の EDI メッセージが必要です。

メッセージの 1 回のパスで、(異なるタイプの) 複数の宛先に複数の分割およびルーティング操作を実行できます。

バグの報告

13.6. SMOOKS による分割メッセージのストリーミング

Smooks によってメッセージをストリーミングすると、次のようになります。

- ルーティングするフラグメントのスタンドアロンメッセージ (分割) を繰り返し作成します。
- 一意の beanId の下で、分割メッセージを Bean コンテキストに繰り返しバインドします。
- 分割メッセージを必要なエンドポイント (ファイル、DB、JMS、または ESB のいずれか) に繰り返しルーティングします。

これらの操作は、ソースメッセージで見つかった分割メッセージの各インスタンスに発生します。たとえば、注文メッセージの orderItem ごとに発生します。

バグの報告

13.7. 分割メッセージの作成方法

- 基本的な (変換/エンリッチされていない) フラグメントの分割とバインド。これにより、メッセージフラグメントが (繰り返し) XML 形式にシリアル化され、Bean コンテキストに文字列として保存されます。
- Java バインドおよびテンプレートカートリッジを使用するより複雑なアプローチでは、Smooks を設定して、ソースメッセージからデータを抽出し、Bean コンテキストに入れ (jb:bean 設定を使用)、テンプレートを適用して (オプション)、分割メッセージを作成できます。これには、次の利点があります。
  - 基本オプションと同様、XML だけでなく、分割フラグメントを変換できます。
  - メッセージのエンリッチを実現します。
  - 複数のソースフラグメントからのデータを各分割メッセージにマージする機能を使用して、より複雑な分割を実現します。たとえば、orderItem フラグメントだけでなく、注文ヘッダー情報も同様です。
  - 分割メッセージとして Java オブジェクトを分割およびルーティングできます (たとえば、JMS 経由)。

バグの報告

13.8. メッセージのシリアル化

1. メッセージのフラグメントを分割およびルーティングするには、ルーティングカートリッジの基本的な frag:serialize および *:router コンポーネント (jms:router、file:router など) を使用します。frag:serialize コンポーネントには、http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd namespace に独自の設定があります。

2. 以下の例を使用して、SOAP メッセージ本文のコンテンツをシリアル化し、soapBody の beanId の下の Bean コンテキストに保存します。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd">

    <frag:serialize fragment="Envelope/Body" bindTo="soapBody" childContentOnly="true"/>

</smooks-resource-list>
```

3. 次のコードを使用して、実行します。

```
Smooks smooks = new Smooks(configStream);
JavaResult javaResult = new JavaResult();

smooks.filterSource(new StreamSource(soapMessageStream), javaResult);

String bodyContent = javaResult.getBean("soapBody").toString().trim();
```

4. これをプログラムで行うには、次のコードを使用します。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new FragmentSerializer().setBindTo("soapBody"), "Envelope/Body");

JavaResult javaResult = new JavaResult();
smooks.filterSource(new StreamSource(soapMessageStream), javaResult);

String bodyContent = javaResult.getBean("soapBody").toString().trim();
```

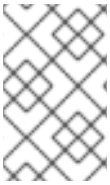
バグの報告

13.9. 分割メッセージのルーティングの例

以下は、分割メッセージ (今回は注文品目のフラグメント) を処理のために JMS 宛先にルーティングするための設定を示す簡単な例です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd"
xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd">

    <!-- Create the split messages for the order items... -->
    <frag:serialize fragment="order-items/order-item" bindTo="orderItem" />

    <!-- Route each order items split mesage to the orderItem JMS processing queue... -->
```

```
    <jms:router routeOnElement="order-items/order-item" beanId="orderItem"
destination="orderItemProcessingQueue" />

</smooks-resource-list>
```



注記

jms:router は、他のルーターの代わりに使用できます。たとえば、JBoss ESB で使用する場合は、esbr:routeBean 設定を使用して、分割メッセージを任意の ESB エンドポイントにルーティングできます。

バグの報告

13.10. ファイルベースのルーティング

ファイルベースのルーティングは、http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd 設定 namespace の file:outputStream 設定によって実行されます。次の Smooks 機能を組み合わせて、メッセージをファイルシステム上の小さなメッセージに分割できます。

バグの報告

13.11. ファイルベースのルーティングコンポーネント

表13.1 ファイルベースのルーティングコンポーネント

コンポーネント	説明
Javabean カートリッジ	メッセージからデータを抽出し、Bean コンテキストの変数に保持します。テンプレートデータモデルとして使用される注文および注文品目データを取得するために、DOM NodeModel を使用することもできます。
file:outputStream	ルーティングカートリッジからのこの設定は、ファイルシステムストリームの管理に使用されます (命名、開始、終了、調整の作成など)。
テンプレートカートリッジ (FreeMarker テンプレート)	Javabean カートリッジによって Bean コンテキストにバインドされたデータから個別の分割メッセージを生成するために使用されます (上記の最初のポイントを参照)。テンプレート結果は、ファイル出力ストリームに書き込まれます (上記の 2 つ目のポイントを参照)。

バグの報告

13.12. 巨大なメッセージの処理

巨大なメッセージの処理

この例では、個別の注文品目の詳細をファイルにルーティングしながら、大量の注文メッセージを送信する必要があります。分割メッセージには、注文ヘッダーとルート要素からのデータが含まれています。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM, so we can process huge messages...
    -->
    <core:filterSettings type="SAX" />

    <!-- Extract and decode data from the message. Used in the freemarker template (below).
         Note that we could also use a NodeModel here... -->
(1)  <jb:bean beanId="order" class="java.util.Hashtable" createOnElement="order">
       <jb:value property="orderId" decoder="Integer" data="order/@id"/>
       <jb:value property="customerNumber" decoder="Long" data="header/customer/@number"/>
       <jb:value property="customerName" data="header/customer"/>
       <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
(2)  <jb:bean beanId="orderItem" class="java.util.Hashtable" createOnElement="order-item">
       <jb:value property="itemId" decoder="Integer" data="order-item/@id"/>
       <jb:value property="productId" decoder="Long" data="order-item/product"/>
       <jb:value property="quantity" decoder="Integer" data="order-item/quantity"/>
       <jb:value property="price" decoder="Double" data="order-item/price"/>
    </jb:bean>

    <!-- Create/open a file output stream. This is writen to by the freemarker template (below).. -->
(3)  <file:outputStream openOnElement="order-item" resourceName="orderItemSplitStream">
       <file:fileNamePattern>order-${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>
       <file:destinationDirectoryPattern>target/orders</file:destinationDirectoryPattern>
       <file:listFileNamePattern>order-${order.orderId}.lst</file:listFileNamePattern>

       <file:highWaterMark mark="10"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this freemarker template,
    outputting the result to the "orderItemSplitStream" OutputStream, which is the file
    output stream configured above.
    -->
(4)  <ftl:freemarker applyOnElement="order-item">
       <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
       <ftl:use>
          <!-- Output the templating result to the "orderItemSplitStream" file output stream... -->
          <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
       </ftl:use>
```

```
        </ftl:freemarker>

</smooks-resource-list>
```

上記の 1 番と 2 番に示されている Smooks リソース設定は、注文ヘッダー情報 (設定番号 1) と注文品目情報 (設定番号 2) を抽出するための Java バインドを定義します。巨大なメッセージを処理する場合は、一度にメモリーにあるものが現在の注文品目のみであることを確認してください。Smooks Javabean Cartridge が、これらすべてを管理し、注文品目フラグメントが処理される際、orderItem Bean を作成および再作成します。

設定番号 3 の **file:outputStream** 設定は、ファイルシステム上のファイルの生成を管理します。設定からわかるように、ファイル名は Bean コンテキストのデータから動的に構成できます。**highWaterMark** 設定パラメーターによってファイルの作成を調整できることもわかります。これにより、ターゲットファイルシステムを圧倒しないように、ファイル作成を管理できます。

Smooks リソース設定番号 4 は、分割メッセージを **file:outputStream** (設定番号 3) によって作成された OutputStream に書き込むために使用される FreeMarker テンプレートリソースを定義します。設定 4 が **file:outputStream** リソースを参照する方法を確認してください。Freemarker テンプレートは次のとおりです。

```
<orderitem id="${.vars["order-item"].@id}" order="${order.@id}">
    <customer>
        <name>${order.header.customer}</name>
        <number>${order.header.customer.@number}</number>
    </customer>
    <details>
        <productId>${.vars["order-item"].product}</productId>
        <quantity>${.vars["order-item"].quantity}</quantity>
        <price>${.vars["order-item"].price}</price>
    </details>
</orderitem>
```

バグの報告

13.13. JMS ルーティング

JMS ルーティング

JMSルーティングは、http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd 設定 namespace の jms:router 設定で実行されます。以下は、orderItem_xml Bean を smooks.exampleQueue という名前の JMS キューにルーティングする jms:router 設定の例です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM, so we can process huge messages...
    -->
    <core:filterSettings type="SAX" />
```

```
(1)    <resource-config selector="order,order-item">
         <resource>org.milyn.delivery.DomModelCreator</resource>
       </resource-config>

(2)    <jms:router routeOnElement="order-item" beanId="orderItem_xml"
destination="smooks.exampleQueue">
         <jms:message>
           <!-- Need to use special FreeMarker variable ".vars" -->
           <jms:correlationIdPattern>${order.@id}-${.vars["order-item"].@id}
</jms:correlationIdPattern>
         </jms:message>
         <jms:highWaterMark mark="3"/>
       </jms:router>

(3)    <ftl:freemarker applyOnElement="order-item">
         <!--
         Note in the template that we need to use the special FreeMarker variable ".vars"
         because of the hyphenated variable names ("order-item"). See
http://freemarker.org/docs/ref_specvar.html.
         -->
         <ftl:template>/orderitem-split.ftl</ftl:template>
         <ftl:use>
           <!-- Bind the templating result into the bean context, from where
           it can be accessed by the JMSRouter (configured above). -->
           <ftl:bindTo id="orderItem_xml"/>
         </ftl:use>
       </ftl:freemarker>

</smooks-resource-list>
```

この場合、FreeMarker テンプレート操作の結果を JMS キューにルーティングします (つまり、文字列として)。完全なオブジェクトモデルをルーティングすることもできます。その場合、シリアル化された ObjectMessage としてルーティングされます。

バグの報告

13.14. データベースへのルーティング

1. 注文および注文品目データをデータベースにルーティングするには、データストリームから注文および注文品目データを抽出する一連の Java バインドを定義する必要があります。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <!-- Extract the order data... -->
    <jb:bean beanId="order" class="java.util.Hashtable" createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
    </jb:bean>

    <!-- Extract the order-item data... -->
```

```
    <jb:bean beanId="orderItem" class="java.util.Hashtable" createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-item/price"/>
    </jb:bean>
```

2. 次に、データソース設定とそのデータソースを使用して、Java オブジェクトモデルにバインドされたデータをデータベースに挿入する多くの db:executor 設定を定義する必要があります。データベースへの直接接続を取得するためのデータソース設定 (namespace http://www.milyn.org/xsd/smooks/datasource-1.3.xsd) です。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-1.3.xsd">

    <ds:direct bindOnElement="#document"
            datasource="DBExtractTransformLoadDS"
            driver="org.hsqldb.jdbcDriver"
            url="jdbc:hsqldb:hsql://localhost:9201/milyn-hsql-9201"
            username="sa"
            password=""
            autoCommit="false" />

</smooks-resource-list>
```

3. データベース接続を取得するために、JNDI データソースを使用できます。

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-1.3.xsd">

    <!-- This JNDI datasource can handle JDBC and JTA transactions or
            it can leave the transaction managment to an other external component.
            An external component could be an other Smooks visitor, the EJB transaction
manager
            or you can do it your self. -->
    <ds:JNDI
        bindOnElement="#document"
        datasource="DBExtractTransformLoadDS"
        datasourceJndi="java:/someDS"
        transactionManager="JTA"
        transactionJndi="java:/mockTransaction"
        targetProfile="jta"/>

</smooks-resource-list>
```

4. データソーススキーマは、データソースを設定する方法を説明および文書化します。これは db:executor 設定です (namespace http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd):

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                    xmlns:db="http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd">
```

```
    <!-- Assert whether it's an insert or update. Need to do this just before we do the
insert/update... -->
    <db:executor executeOnElement="order-items" datasource="DBExtractTransformLoadDS"
executeBefore="true">
        <db:statement>select OrderId from ORDERS where OrderId = ${order.orderId}
</db:statement>
        <db:resultSet name="orderExistsRS"/>
    </db:executor>

    <!-- If it's an insert (orderExistsRS.isEmpty()), insert the order before we process the order
items... -->
    <db:executor executeOnElement="order-items" datasource="DBExtractTransformLoadDS"
executeBefore="true">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>INSERT INTO ORDERS VALUES(${order.orderId},
${order.customerNumber}, ${order.customerName})</db:statement>
    </db:executor>

    <!-- And insert each orderItem... -->
    <db:executor executeOnElement="order-item" datasource="DBExtractTransformLoadDS"
executeBefore="false">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>INSERT INTO ORDERITEMS VALUES (${orderItem.itemId},
${order.orderId}, ${orderItem.productId}, ${orderItem.quantity}, ${orderItem.price})
</db:statement>
    </db:executor>

    <!-- Ignoring updates for now!! -->

</smooks-resource-list>
```

バグの報告

第14章 SMOOKS の拡張

14.1. SMOOKS の API

API

既存のすべての Smooks 機能 (Java バインド、EDI 処理など) は、明確に定義された多くの API の拡張によって構築されています。

Smooks の主要な拡張ポイント/API は、リーダー API とビジター API です。

リーダー API

ソース/入力データ (リーダー) を処理して、すべてのメッセージフラグメントとサブフラグメントに明確に定義された一連の階層イベント (SAX イベントモデルに基づく) として他の Smooks コンポーネントによって消費できるようにするためのもの。

ビジター API

ソース/入力リーダーによって生成されたメッセージフラグメント SAX イベントを消費するためのもの。

バグの報告

14.2. SMOOKS コンポーネントの設定

すべての Smooks コンポーネントは、まったく同じ方法で設定されます。Smooks Core コードを使用する場合、すべての Smooks コンポーネントは、**SmooksResourceConfiguration** インスタンスを使用して設定された **resources** です。

バグの報告

14.3. NAMESPACE 固有の設定

Smooks は、コンポーネントの namespace (XSD) 固有の XML 設定を構成する機能を提供します。最も基本的な設定 (および **SmooksResourceConfiguration** クラスに直接マッピングされる設定) は、基本設定 namespace (http://www.milyn.org/xsd/smooks-1.1.xsd)。

バグの報告

14.4. NAMESPACE 固有の設定例

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <resource-config selector="">
        <resource></resource>
        <param name=""></param>
    </resource-config>

</smooks-resource-list>
```

- **selector** 属性は、リソースを選択する機能です (たとえば、ビジター実装の XPath にすることができます)。
- **resource** 要素は実際のリソースです。これは、Java クラス名またはテンプレートなどの他の形式のリソースにすることができます。このセクションの残りの部分では、リソースは Java クラス名であると想定されます。
- **param** 要素は、resource 要素に定義されたリソースの設定パラメーターです。

バグの報告

14.5. ランタイム表現

Smooks は、リソースの実行時表現を作成するためのすべての詳細 (たとえば、リソース要素に指定されたクラスの構成) を処理し、すべての設定パラメーターを注入します。また、リソースタイプとは何か、およびそこからセレクターなどを解釈する方法も明らかにします。(たとえば、リソースがビジターインスタンスである場合は、ソースメッセージフラグメントを選択するセレクターが XPath であることを認識します。)

バグの報告

14.6. 設定アノテーション

コンポーネントが作成されたら、**<param>** 要素の詳細を設定する必要があります。これは、**@ConfigParam** および **@Config** アノテーションを使用すると、実行されます。

バグの報告

14.7. @CONFIGPARAM アノテーション

@ConfigParam アノテーションは、アノテーション付きのプロパティ自身と同じ名前を持つ **<param>** 要素から名前付きパラメーターを反射的に注入します。名前は異なる場合がありますが、デフォルトの動作はコンポーネントプロパティーの名前に一致します。

バグの報告

14.8. @CONFICPARAM の利点

このアノテーションにより、次の理由でコンポーネントから余分なコードが削除されます。

- <param> 値をアノテーション付きのコンポーネントプロパティーに設定する前に、そのデコードを処理します。Smooks は、すべての主要な型 (int、Double、File、Enum など) に DataDecoder を提供しますが、すぐに使用できるデコーダーが特定のデコード要件をカバーしていない場合は、カスタム DataDecoder を実装および使用できます (例: **@ConfigParam(decoder = MyQuirkyDataDecoder.class)**)。カスタムデコーダーが登録されている場合、Smooks はカスタムデコーダーを自動的に使用します (つまり、このアノテーショ

ンでデコーダープロパティーを定義する必要はありません)。Smooks が、特定のデータ型をデコードするために、DataDecoder 実装を自動的に見つけられるように、DataDecoder 実装を登録する方法の詳細については、DataDecoder Javadocs を参照してください。

- **config** プロパティの **choice** 制約をサポートし、設定値が定義された選択値の 1 つではない場合、設定例外を生成します。たとえば、**ON** と **OFF** の制約付きの値セットを持つプロパティーがあるとします。このアノテーションの Choice プロパティーを使用して、設定を制約したり、例外を発生させたりすることができます。(たとえば、**@ConfigParam(choice = {"ON", "OFF"})** です。)
- **@ConfigParam(defaultVal = "true")** など、デフォルトの設定値を指定できます。
- **@ConfigParam(use = Use.OPTIONAL)** など、プロパティー設定値が必須かオプションかを指定できます。デフォルトでは、すべてのプロパティーは、**REQUIRED** ですが、**defaultVal** を設定すると、プロパティーは、**OPTIONAL** と暗黙的にマークされます。

バグの報告

14.9. @CONFIGPARAM アノテーションの使用

この例は、アノテーション付きのコンポーネント **DataSeeder** とそれに対応する Smooks 設定を示しています。

```
public class DataSeeder
{
   @ConfigParam
   private File seedDataFile;

   public File getSeedDataFile()
   {
      return seedDataFile;
   }

   // etc...
}
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
   <resource-config selector="dataSeeder">
      <resource>com.acme.DataSeeder</resource>
         <param name="seedDataFile">./seedData.xml</param>
      </resource-config>
</smooks-resource-list>
```

バグの報告

14.10. @CONFIG アノテーション

@Config アノテーションは、コンポーネントリソースに関連付けられた完全な **SmooksResourceConfiguration** インスタンスをアノテーション付きのコンポーネントプロパティに反射的に注入します。タイプ **SmooksResourceConfiguration** ではないコンポーネントプロパティーにこのアノテーションを追加すると、エラーが発生します。

バグの報告

14.11. @CONFIG アノテーションの使用

```
public class MySmooksComponent
{
    @Config
    private SmooksResourceConfiguration config;

    // etc...
```

バグの報告

14.12. @INITIALIZE と @UNINITIALIZE

場合によっては、コンポーネントに初期化コードを記述する必要がある複雑な設定が必要です。このために、Smooks は **@Initialize** アノテーションを提供しています。

同様に、関連する Smooks インスタンスが破棄される (ガベージコレクション) 際、初期化中に行った操作を元に戻す必要がある場合があります。たとえば、初期化中に取得したリソースを解放する場合などです。このために、Smooks は **@Uninitialize** アノテーションを提供しています。

バグの報告

14.13. 基本的な初期化/初期化解除シーケンス

これは、基本的な初期化/初期化解除シーケンスです。

```
smooks = new Smooks(..);

// Initialize all annotated components
@Initialize

// Use the smooks instance through a series of filterSource invocations...
smooks.filterSource(...);
smooks.filterSource(...);
smooks.filterSource(...);
... etc ...

smooks.close();

// Uninitialize all annotated components
@Uninitialize
```

バグの報告

14.14. @INITIALIZE と @UNINITIALIZE の使用

概要

この例では、初期化時にデータベースへの複数の接続を開いて、Smooks インスタンスを閉じる際、それらのデータベースリソースをすべて解放する必要があるコンポーネントがあるとします。

```
public class MultiDataSourceAccessor
{
    @ConfigParam
    private File dataSourceConfig;

    Map<String, Datasource> datasources = new HashMap<String, Datasource>();

    @Initialize
    public void createDataSources()
    {
        // Add DS creation code here....
        // Read the dataSourceConfig property to read the DS configs...
    }

    @Uninitialize
    public void releaseDataSources()
    {
        // Add DS release code here....
    }

    // etc...
}
```

上記の **@Initialize** および **@Uninitialize** アノテーションを使用する場合は、次の点に注意する必要があります。

- **@Initialize** および **@Uninitialize** メソッドは、引数なしのパブリックメソッドである必要があります。
- **@ConfigParam** プロパティーは、最初の **@Initialize** メソッドが呼び出される前に、すべて初期化されます。したがって、**@ConfigParam** コンポーネントプロパティーを初期化プロセスへの入力として使用できます。
- **@Uninitialize** メソッドは、すべて **Smooks.close** メソッドの呼び出しに応答して、呼び出されます。

バグの報告

14.15. カスタム設定 NAMESPACE の定義

Smooks は、コンポーネントのカスタム設定 namespace を定義する機能をサポートしています。これにより、**<resource-config>** 基本設定を使用して、検証可能なコンポーネントの XSD ベースのカスタム設定をすべて一般的な Smooks リソースとして扱うのではなく、サポートできます。

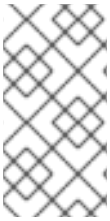
バグの報告

14.16. カスタム設定 NAMESPACE の使用

基本的なプロセスには、次の 2 つの手順が含まれます。

1. 基本 http://www.milyn.org/xsd/smooks-1.1.xsd 設定 namespace を拡張するコンポーネントの設定 XSD を記述します。この XSD は、コンポーネントのクラスパスで提供する必要があります。**/META-INF/** フォルダーに配置され、namespace URI と同じパスを持つ必要があります。たとえば、拡張 namespace URI が http://www.acme.com/schemas/smooks/acme-core-1.0.xsd の場合は、**/META-INF/schemas/smooks/acme-core-1.0.xsd** のクラスパスに物理 XSD ファイルを指定する必要があります。

2. カスタム namespace 設定を **SmooksResourceConfiguration** インスタンスにマッピングする Smooks 設定 namespace マッピング設定ファイルを記述します。このファイルは、マッピングする namespace の名前に基づいて (慣例により)、名前を付ける必要があり、XSD と同じフォルダー内のクラスパスに物理的に配置する必要があります。上記の例を拡張すると、Smooks マッピングファイルは、**/META-INF/schemas/smooks/acme-core-1.0.xsd-smooks.xml** になります。**-smooks.xml** 接尾辞に注意してください。



注記

この機能に慣れるための最も簡単な方法は、Smooks コード自体の既存の拡張 namespace 設定を確認することです。すべての Smooks コンポーネント (Java バインド機能を含む) は、この機能を使用して、設定を定義します。Smooks Core 自体は、多くの拡張設定 namespace を定義します。

バグの報告

14.17. ソースリーダーの実装

カスタムデータ形式のソースリーダーを実装すると、Java バインド、テンプレート、保持、検証、分割、ルーティングなど、そのデータ形式に対するすべての Smooks 機能をすぐに開くことができます。Smooks の唯一の要件は、リーダーが、Java JDK からの標準の **org.xml.sax.XMLReader** インターフェイスを実装していることです。ただし、リーダー実装を設定できるようにする場合は、リーダーが、**org.milyn.xml.SmooksXMLReader** インターフェースを実装する必要があります。**org.milyn.xml.SmooksXMLReader** は **org.xml.sax.XMLReader** の拡張です。既存の **org.xml.sax.XMLReader** 実装を使用することも、新しいものを実装することも簡単です。

詳細については、http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/XMLReader.html を参照してください。

バグの報告

14.18. SMOOKS で使用するソースリーダーの実装

1. 次のように、まず、基本的なリーダークラスを実装する必要があります。

```
public class MyCSVReader implements SmooksXMLReader
{
  // Implement all of the XMLReader methods...
}
```

特に興味深いものは、**org.xml.sax.XMLReader** インターフェイスの 2 つのメソッドです。

1. **setContentHandler(ContentHandler)** は Smooks Core によって呼び出されます。リーダーの **org.xml.sax.ContentHandler** インスタンスを設定します。**org.xml.sax.ContentHandler** インスタンスメソッドは、**parse(InputSource)** メソッ

ド内から呼び出されます。

2. **parse(InputSource)**: これは、ソースデータ入力ストリームを受け取り、解析し (つまり、この例の場合は、CSVストリーム)、**setContentHandler(ContentHandler)** メソッドで提供される **org.xml.sax.ContentHandler** インスタンスの呼び出しで SAX イベントストリームを生成するメソッドです。

詳細については、http://download.oracle.com/javase/6/docs/api/org/xml/sax/ContentHandler.html を参照してください。

2. CSV レコードに関連付けられたフィールドの名前を使用して、CSV リーダーを設定します。カスタムリーダー実装の設定は、いずれの Smooks コンポーネントでも同じです。以下の例を参照してください。

```
public class MyCSVReader implements SmooksXMLReader
{

    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the "fields" <param> on the reader
config.

    public void setContentHandler(ContentHandler contentHandler) {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws IOException, SAXException {
        // TODO: Implement parsing of CSV Stream...
    }

    // Other XMLReader methods...
}
```

3. 基本的なリーダー実装スタブができると、単体テストの作成を開始して、新しいリーダー実装をテストできます。これを行うには、CSV 入力が必要です。**names.csv** という名前のファイル内の名前の単純なリストを特徴とする以下の例を確認してください。

```
Tom,Jones
Mike,Jones
Mark,Jones
```

4. テスト Smooks 設定を使用して、MyCSVReader で Smooks を設定します。前述のとおり、Smooks では、すべてがリソースであり、基本的な **<resource-config>** 設定を行うことができます。これは、問題なく動作しますが、少し「うるさい」ため、Smooks は、特にリーダーを設定するために、基本的な **<reader>** 設定要素を提供しています。テストの設定は、**mycsvread-config.xml** で次のようになります。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
  <reader class="com.acme.MyCSVReader">
    <params>
      <param name="fields">firstname,lastname</param>
```

```
      </params>
    </reader>
</smooks-resource-list>
```

5. JUnit テストクラスを実装します。

```
public class MyCSVReaderTest extends TestCase
{

    public void test() {
        Smooks smooks = new Smooks(getClass().getResourceAsStream("mycsvread-
config.xml"));
        StringResult serializedCSVEvents = new StringResult();

        smooks.filterSource(new
StreamSource(getClass().getResourceAsStream("names.csv")), serializedCSVEvents);

        System.out.println(serializedCSVEvents);

        // TODO: add assertions etc
    }
}
```

6. **parse** メソッドを実装します。

```
public class MyCSVReader implements SmooksXMLReader
{
    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the "fields" <param> on the reader
config.

    public void setContentHandler(ContentHandler contentHandler)
    {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws IOException, SAXException
    {
        BufferedReader csvRecordReader = new
BufferedReader(csvInputSource.getCharacterStream());
        String csvRecord;

        // Send the start of message events to the handler...
        contentHandler.startDocument();
        contentHandler.startElement(XMLConstants.NULL_NS_URI, "message-root", "", new
AttributesImpl());

        csvRecord = csvRecordReader.readLine();
        while(csvRecord != null)
        {
            String[] fieldValues = csvRecord.split(",");

            // perform checks...
```

```
            // Send the events for this record...
            contentHandler.startElement(XMLConstants.NULL_NS_URI, "record", "", new
AttributesImpl());
            for(int i = 0; i < fields.length; i++)
            {
                contentHandler.startElement(XMLConstants.NULL_NS_URI, fields[i], "", new
AttributesImpl());
                contentHandler.characters(fieldValues[i].toCharArray(), 0, fieldValues[i].length());
                contentHandler.endElement(XMLConstants.NULL_NS_URI, fields[i], "");
            }
            contentHandler.endElement(XMLConstants.NULL_NS_URI, "record", "");

            csvRecord = csvRecordReader.readLine();
        }

        // Send the end of message events to the handler...
        contentHandler.endElement(XMLConstants.NULL_NS_URI, "message-root", "");
        contentHandler.endDocument();
    }

    // Other XMLReader methods...
}
```

7. 単体テストクラスを実行して、コンソールに次の出力を表示します (フォーマット済み)。

```
<message-root>
    <record>
        <firstname>Tom</firstname>
        <lastname>Jones</lastname>
    </record>
    <record>
        <firstname>Mike</firstname>
        <lastname>Jones</lastname>
    </record>
    <record>
        <firstname>Mark</firstname>
        <lastname>Jones</lastname>
    </record>
</message-root>
```

この後は、テストの拡張、リーダー実装コードの強化などのケースです。その後、リーダーを使用して、Smooks でサポートされているさまざまな操作を実行できます。

バグの報告

14.19. JAVA-BINDING-CONFIG.XML を使用したリーダーの設定例

次の設定 (**java-binding-config.xml**) を使用すると、名前を **PersonName** オブジェクトの**List** にバインドできます。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">
```

```
    <reader class="com.acme.MyCSVReader">
        <params>
         <param name="fields">firstname,lastname</param>
        </params>
    </reader>
    <jb:bean beanId="peopleNames" class="java.util.ArrayList" createOnElement="message-root">
        <jb:wiring beanIdRef="personName" />
    </jb:bean>
    <jb:bean beanId="personName" class="com.acme.PersonName" createOnElement="message-
root/record">
        <jb:value property="first" data="record/firstname" />
        <jb:value property="last" data="record/lastname" />
    </jb:bean>
</smooks-resource-list>
```

その設定のテストは次のとおりです。

```
public class MyCSVReaderTest extends TestCase
{
    public void test_java_binding()
    {
        Smooks smooks = new Smooks(getClass().getResourceAsStream("java-binding-config.xml"));
        JavaResult javaResult = new JavaResult();

        smooks.filterSource(new StreamSource(getClass().getResourceAsStream("names.csv")),
javaResult);

        List<PersonName> peopleNames = (List<PersonName>) javaResult.getBean("peopleNames");

        // TODO: add assertions etc
    }
}
```

バグの報告

14.20. リーダーを使用するためのヒント

- リーダーインスタンスが同時に使用されることはありません。Smooks Coreは、メッセージごとに新しいインスタンスを作成するか、**readerPoolSizeFilterSettings** プロパティに従って、インスタンスをプールおよび再利用します。
- リーダーは、現在のフィルタリングコンテキストの Smooks **ExecutionContext** にアクセスする必要がある場合、**org.milyn.xml.SmooksXMLReader** インターフェイスを実装する必要があります。
- ソースデータがバイナリーデータストリームである場合、リーダーは、**org.milyn.delivery.StreamReader** インターフェイスを実装する必要があります。
- **GenericReaderConfigurator** インスタンスを使用して、ソースコード (単体テストなど) 内でリーダーを設定し、**Smooks** インスタンスに設定できます。
- 基本的な <reader> 設定は問題ありませんが、カスタム CSV リーダー実装用にカスタム設定 namespace (XSD) を定義できます。このトピックは、ここでは扱いません。ソースコードを確認して、Smooks に付属するリーダー実装の拡張設定 namespace

(**EDIReader**、**CSVReader**、**JSONReader** など) を確認します。これから、独自のカスタムリーダーでこれを行う方法を考え出すことができるはずです。

バグの報告

14.21. バイナリーソースリーダー

*バイナリーソースリーダー*はバイナリーデータソースのリーダーです。リーダーは、**org.milyn.delivery.StreamReader** インターフェイスを実装する必要があります。これは、**InputStream** が提供されるように、Smooks ランタイムに指示する単なるマーカーインターフェイスです。

バイナリーリーダーの実装は、非バイナリーリーダーの実装 (上記を参照) と本質的に同じですが、**parse** メソッドの実装が、**InputSource** からの**InputStream** を使用し (つまり、**InputSource.getCharacterStream()** ではなく、**InputSource..getByteStream()** を呼び出し)、デコードされたバイナリーデータから XML イベントを生成する必要がある点では異なります。

バグの報告

14.22. バイナリーソースリーダーの実装

1. バイナリーソースリーダーを実装するには、次の **parse** メソッド実装を確認してください。

```
public static class BinaryFormatXXReader implements SmooksXMLReader, StreamReader
{
    @ConfigParam
    private String xProtocolVersion;

    @ConfigParam
    private int someOtherXProtocolConfig;

    // etc...

    public void parse(InputSource inputSource) throws IOException, SAXException {
        // Use the InputStream (binary) on the InputSource...
        InputStream binStream = inputSource.getByteStream();

        // Create and configure the data decoder...
        BinaryFormatXDecoder xDecoder = new BinaryFormatXDecoder();
        xDecoder.setProtocolVersion(xProtocolVersion);
        xDecoder.setSomeOtherXProtocolConfig(someOtherXProtocolConfig);
        xDecoder.setXSource(binStream);

        // Generate the XML Events on the contentHandler...
        contentHandler.startDocument();

        // Use xDecoder to fire startElement, endElement etc events on the contentHandler (see
previous section)...

        contentHandler.endDocument();
    }
```

```
    // etc....
}
```

2. 他のリーダーと同様、Smooks 設定に **BinaryFormatXXReader** リーダーを設定します。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <reader class="com.acme.BinaryFormatXXReader">
        <params>
         <param name="xProtocolVersion">2.5.7</param>
         <param name="someOtherXProtocolConfig">1</param>
            ... etc...
        </params>
    </reader>

    ... Other Smooks configurations e.g. <jb:bean> configs for binding the binary data into Java objects...

</smooks-resource-list>
```

3. Smooks の実行コードを実行します (**StreamSource** に提供される **InputStream** に注意してください)。この場合、XML オブジェクトと Java オブジェクトの 2 つの結果が生成されます。

```
StreamResult xmlResult = new StreamResult(xmlOutWriter);
JavaResult javaResult = new JavaResult();

InputStream xBinaryInputStream = getXByteStream();

smooks.filterSource(new StreamSource(xBinaryInputStream), xmlResult, javaResult);

// etc... Use the beans in the javaResult...
```

バグの報告

14.23. ビジター実装

ビジター実装 は、Smooks の主要な機能です。Smooks のすぐに使える機能 (Java バインド、テンプレート、保持など) のほとんどは、1 つ以上のビジター実装を使用して、作成されました。ビジター実装は、多くの場合、**ExecutionContext** および **ApplicationContext** コンテキストオブジェクトでコラボレーションして、共通の目標を達成します。

バグの報告

14.24. サポート対象のビジター実装

1. **org.milyn.delivery.sax.SAXVisitor** サブインターフェイスに基づく SAX ベースの実装。
2. **org.milyn.delivery.dom.DOMVisitor** サブインターフェイスに基づく DOM ベースの実装。

バグの報告

14.25. SAX と DOM のビジター実装

実装は SAX と DOM の両方をサポートできますが、Red Hat は SAX のみのビジターを実装することを推奨します。通常、SAX ベースの実装は作成が簡単であり、実行も高速です。

重要

すべてのビジター実装は、ステートレスオブジェクトとして扱われます。1 つのビジターインスタンスは、複数のメッセージ、つまり **Smooks.filterSource** メソッドの複数の同時呼び出しで同時に使用できる必要があります。現在の **Smooks.filterSource** 実行に関連するすべての状態は、**ExecutionContext** に保存する必要があります。

バグの報告

14.26. SAX ビジター API

SAX ビジター API は多くのインターフェイスで構成されています。これらのインターフェイスは、**SAXVisitor** 実装が取得および処理できる **org.xml.sax.ContentHandler** SAX イベントに基づいています。**SAXVisitor** 実装によって解決されるユースケースに応じて、これらのインターフェイスの 1 つまたはすべてを実装する必要がある場合があります。

バグの報告

14.27. SAX ビジター API インターフェイス

org.milyn.delivery.sax.SAXVisitBefore

対象のフラグメント要素の **startElement** SAX イベントを取得します。

```
public interface SAXVisitBefore extends SAXVisitor
{
    void visitBefore(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException;
}
```

org.milyn.delivery.sax.SAXVisitChildren

対象のフラグメント要素の文字ベースの SAX イベント、および子フラグメント要素の **startElement** イベントに対応する Smooks 生成 (疑似) イベントを取得します。

```
public interface SAXVisitChildren extends SAXVisitor
{
    void onChildText(SAXElement element, SAXText childText, ExecutionContext
                executionContext) throws SmooksException, IOException;

    void onChildElement(SAXElement element, SAXElement childElement,
        ExecutionContext executionContext) throws SmooksException, IOException;
}
```

org.milyn.delivery.sax.SAXVisitAfter

対象のフラグメント要素の **endElement** SAX イベントを取得します。

```
 public interface SAXVisitAfter extends SAXVisitor
{
    void visitAfter(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException;
}
```

バグの報告

14.28. SAX ビジター API の例

XML を使用した API イベントの図解

これにより、3 つのインターフェイスが **org.milyn.delivery.sax.SAXElementVisitor** インターフェイス内で 1 つのインターフェイスにまとめられます。

```
<message>
   <target-fragment>      <!-- SAXVisitBefore.visitBefore -->
      Text!!              <!-- SAXVisitChildren.onChildText -->
      <child>             <!-- SAXVisitChildren.onChildElement -->
      </child>
   </target-fragment>     <!-- SAXVisitAfter.visitAfter -->
</message>
```

上記は、XML としてのソースメッセージイベントストリームの図です。EDI、CSV、JSON、またはその他の形式の可能性があります。これは、読みやすいように、XML としてシリアル化されたソースメッセージイベントストリームであると考えてください。

上記の SAX インターフェイスからわかるように、**org.milyn.delivery.sax.SAXElement** タイプは、すべてのメソッド呼び出しで渡されます。このオブジェクトには、属性とその値を含む、対象のフラグメント要素に関する詳細が含まれています。また、テキストの蓄積を管理するためのメソッド、および **Smooks.filterSource(Source, Result)** メソッド呼び出しで渡された可能性がある **StreamResult** インスタンスに関連する **Writer** へのアクセスが含まれています。

バグの報告

14.29. SAX によるテキスト蓄積

SAX はストリームベースの処理モデルです。ドキュメントオブジェクトモデル (DOM) を作成したり、イベントデータを蓄積したりすることはありません。これが、巨大なメッセージストリームの処理に適した処理モデルである理由です。

バグの報告

14.30. ORG.MILYN.DELIVERY.SAX.SAXELEMENT

org.milyn.delivery.sax.SAXElement は、常に対象の要素に関連する属性データを含みますが、フラグ

メントの子テキストデータを含みません。SAX イベント (**SAXVisitChildren.onChildText**) が **SAXVisitBefore.visitBefore** イベントと **SAXVisitAfter.visitAfter** イベントの間に発生します。テキストイベントは、**SAXElement** に蓄積されません。その結果、パフォーマンスが大幅に低下する可能性があるためです。この欠点は、**SAXVisitor** 実装がフラグメントのテキストコンテンツにアクセスする必要がある場合、対象のフラグメントのテキストを蓄積するように、Smooks に明示的に指示する必要があることです。これは、**SAXVisitor** の **SAXVisitBefore.visitBefore** メソッド実装内から **SAXElement.accumulateText** メソッドを呼び出すと、実行されます。

バグの報告

14.31. テキスト蓄積の例

```
public class MyVisitor implements SAXVisitBefore, SAXVisitAfter
{
  public void visitBefore(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    element.accumulateText();
  }

  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    String fragmentText = element.getTextContent();

    // ... etc ...
  }
}
```

バグの報告

14.32. @TEXTCONSUMER アノテーション

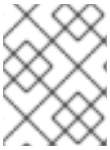
@TextConsumer アノテーションは、**SAXVisitBefore.visitBefore** メソッドを使用する代わりに、**SAXVisitor** 実装をアノテーションするために、使用することができます。

バグの報告

14.33. @TEXTCONSUMER の例

```
@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    String fragmentText = element.getTextContent();
```

```
        // ... etc ...
    }
}
```



注記

フラグメントテキストの全容は、**SAXVisitAfter.visitAfter** イベントの終了後、公開されます。

バグの報告

14.34. STREAMRESULT の書き込み/シリアル化

概要

Smooks.filterSource(Source, Result) メソッドは、多くの異なる **Result** 型実装のうちの 1 つ以上を取ることができ、そのうちの 1 つが **javax.xml.transform.stream.StreamResult** クラスです。Smooks は、**StreamResult** インスタンスでソースを流入させ、再び流出させます。

デフォルトでは、Smooks は常に完全なソースイベントストリームを XML として、**Smooks.filterSource(Source, Result)** メソッドに提供される任意の **StreamResult** インスタンスにシリアル化します。**Smooks.filterSource(Source, Result)** メソッドに提供されたソースが XML ストリームであり、**StreamResult** インスタンスが **Result** インスタンスの 1 つとして提供された場合は、Smooks インスタンスが 1 つ以上のフラグメントを変更する 1 つ以上の **SAXVisitor** 実装で設定されていないかぎり、ソース XML は未変更の **StreamResult** に書き出されます。

バグの報告

14.35. STREAMRESULT の書き込み/シリアル化の設定

1. デフォルトのシリアル化動作をオンまたはオフにするには、フィルター設定にアクセスし、そのように設定します。

2. メッセージフラグメントの 1 つのシリアル化された形式を変更するには、**SAXVisitor** を実装して、変換を実行し、XPath のような式を使用して、メッセージフラグメントをターゲットとします。

3. メッセージフラグメントのシリアル化された形式を変更するには、提供されているテンプレートコンポーネントのいずれかを使用します。これらのコンポーネントは、**SAXVisitor** 実装でもあります。

バグの報告

14.36. SAXVISITOR の実装

1. フラグメントのシリアル化された形式を変換するための **SAXVisitor** を実装するには、**SAXVisitor** 実装が **StreamResult** に書き込まれるように、Smooks をプログラミングします。これは、Smooks が 1 つのフラグメントで複数の **SAXVisitor** 実装のターゲティングをサポートしているためですが、フラグメントごとに **StreamResult** に書き込むことができる **SAXVisitor** は、1 つだけです。

2. 2 つ目の **SAXVisitor** が **StreamResult** に書き込もうとすると、**SAXWriterAccessException** が発生し、Smooks 設定を変更する必要があります。

3. 記述する **StreamResult** を指定するには、**SAXVisitor** が、**StreamResult** への **Writer** の所有権を取得する必要があります。これは、**SAXVisitBefore.visitBefore** メソッド実装内から、**SAXElement.getWriter(SAXVisitor)** メソッドを呼び出し、**this** を **SAXVisitor** パラメーターとして渡すと、実行されます。

バグの報告

14.37. SAXVISITOR の実装例

```
public class MyVisitor implements SAXElementVisitor
{
  public void visitBefore(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    Writer writer = element.getWriter(this);

    // ... write the start of the fragment...
  }

  public void onChildText(SAXElement element, SAXText childText,
                          ExecutionContext executionContext)
                            throws SmooksException, IOException
  {
    Writer writer = element.getWriter(this);

    // ... write the child text...
  }

  public void onChildElement(SAXElement element, SAXElement childElement,
                          ExecutionContext executionContext)
                            throws SmooksException, IOException
  {

  }

  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    Writer writer = element.getWriter(this);
    // ... close the fragment...
  }
}
```

バグの報告

14.38. SAXELEMENT.SETWRITER

SAXElement.setWriter を使用すると、**Writer** インスタンスをリセットするために必要なサブフラグメントのシリアル化を制御できるため、サブフラグメントのシリアル化を流用できます。

シリアル化/変換しているターゲットフラグメントにサブフラグメントがまったくないことがわかっている場合があります。この状況では、**Writer** の所有権を獲得するための **SAXElement.getWriter** メソッドを呼び出すためだけに、**SAXVisitBefore.visitBefore** メソッドを実装することは非効率的です。このため、**@StreamResultWriter** アノテーションがあります。**@TextConsumer** アノテーションと組み合わせて使用すると、**SAXVisitAfter.visitAfter** メソッドを実装するだけで十分です。

バグの報告

14.39. STREAMRESULTWRITER の例

```
@StreamResultWriter
public class MyVisitor implements SAXVisitAfter
{
  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException
  {
    Writer writer = element.getWriter(this);

    // ... serialize to the writer ...
  }
}
```

バグの報告

14.40. SAXTOXMLWRITER

Smooks は、**SAXToXMLWriter** クラスを提供します。**SAXElement** データを XML としてシリアル化するプロセスを簡素化します。このクラスでは、**SAXVisitor** 実装を記述できます。

バグの報告

14.41. SAXTOXMLWRITER の例

```
@StreamResultWriter
public class MyVisitor implements SAXElementVisitor
{
  private SAXToXMLWriter xmlWriter = new SAXToXMLWriter(this, true);

  public void visitBefore(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException
  {
    xmlWriter.writeStartElement(element);
  }

  public void onChildText(SAXElement element, SAXText childText, ExecutionContext
                        executionContext) throws SmooksException, IOException
  {
    xmlWriter.writeText(childText, element);
  }
```

```
    public void onChildElement(SAXElement element, SAXElement childElement,
      ExecutionContext executionContext) throws SmooksException, IOException
    {

    }

    public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException
    {
      xmlWriter.writeEndElement(element);
    }
  }
```

バグの報告

14.42. SAXTOXMLWRITER の設定

1. **SAXToXMLWriter** で **SAXVisitor** 実装を記述する場合は、**SAXToXMLWriter** コンストラクターにブール値を設定します。**encodeSpecialChars** argであり、**rewriteEntities** フィルター設定に基づいて、設定する必要があります。

2. **@StreamResultWriter** アノテーションをクラスから **SAXToXMLWriter** インスタンス宣言に移動します。この結果、Smooks は、**SAXToXMLWriter** インスタンスを作成し、関連する Smooks インスタンスの **rewriteEntities** フィルター設定で初期化されます。

```
@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
  @StreamResultWriter
  private SAXToXMLWriter xmlWriter;

  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                        throws SmooksException, IOException
  {
    xmlWriter.writeStartElement(element);
    xmlWriter.writeText(element);
    xmlWriter.writeEndElement(element);
  }
}
```

バグの報告

14.43. ビジター設定

SAXVisitor 設定は、テスト目的に役立ち、他の Smooks コンポーネントとまったく同様に機能します。Smooks ビジターインスタンスを設定する場合は、設定 **selector** は、XPath 式と同様に解釈されます。ビジターインスタンスは、Smooks インスタンスのプログラムコード内で設定できます。

バグの報告

14.44. ビジター設定の例

この例では、次のような **SAXVisitor** 実装を使用します。

```
@TextConsumer
public class ChangeItemState implements SAXVisitAfter
{
  @StreamResultWriter
  private SAXToXMLWriter xmlWriter;

  @ConfigParam
  private String newState;

  public void visitAfter(SAXElement element, ExecutionContext executionContext)
                                    throws SmooksException, IOException
  {
    element.setAttribute("state", newState);

    xmlWriter.writeStartElement(element);
    xmlWriter.writeText(element);
    xmlWriter.writeEndElement(element);
  }
}
```

ステータスが **OK** の<order-item> フラグメントをトリガーするように、 **ChangeItemState** を宣言的に設定すると、次のようになります。

```
<smooks-resource-list
  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

  <resource-config selector="order-items/order-item[@status = 'OK']">
    <resource>com.acme.ChangeItemState </resource>
    <param name="newState">COMPLETED</param>
  </resource-config>

</smooks-resource-list>
```

カスタム設定 namespace を使用すると、よりクリーンな厳密に型指定された設定を **ChangeItemState** コンポーネントに定義できます。カスタム設定 namespace コンポーネントは次のように設定されます。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
  xmlns:order="http://www.acme.com/schemas/smooks/order.xsd">

  <order:changeItemState itemElement="order-items/order-item[@status = 'OK']"
    newState="COMPLETED" />

</smooks-resource-list>
```

このビジターは、次のように、ソースコードに設定することもできます。

```
Smooks smooks = new Smooks();

smooks.addVisitor(new ChangeItemState().setNewState("COMPLETED"),
```

```
                                "order-items/order-item[@status = 'OK']");

smooks.filterSource(new StreamSource(inReader), new StreamResult(outWriter));
```

バグの報告

14.45. EXECUTIONLIFECYCLECLEANABLE

ExecutionLifecycleCleanable ライフサイクルインターフェイスを実装するビジターコンポーネントはポスト **Smooks.filterSource** ライフサイクル操作を実行できます。以下の例を参照してください。

```
public interface ExecutionLifecycleCleanable extends Visitor
{
   public abstract void executeExecutionLifecycleCleanup(
                                    ExecutionContext executionContext);
}
```

基本的な呼び出しシーケンスは、次のように記述できます (**executeExecutionLifecycleCleanup** 呼び出しに注意)。

```
smooks = new Smooks(..);

smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
 ... etc ...
```

このライフサイクルメソッドでは、**Smooks.filterSource** 実行ライフサイクルの周囲にスコープされたリソースを関連する **ExecutionContext** のためにクリーンアップできます。

バグの報告

14.46. VISITLIFECYCLECLEANABLE

VisitLifecycleCleanable ライフサイクルインターフェイスを実装するビジターコンポーネントはポスト **SAXVisitAfter.visitAfter** ライフサイクル操作を実行できます。

```
public interface VisitLifecycleCleanable extends Visitor
{
   public abstract void executeVisitLifecycleCleanup(ExecutionContext executionContext);
}
```

基本的な呼び出しシーケンスは、次のように記述できます (**executeVisitLifecycleCleanup** 呼び出しに注意)。

```
smooks.filterSource(...);

    <message>
```

```
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!              < --- VisitorXX.onChildText
            <child>             < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!              < --- VisitorXX.onChildText
            <child>             < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
    </message>
    VisitorXX.executeExecutionLifecycleCleanup

smooks.filterSource(...);

    <message>
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!              < --- VisitorXX.onChildText
            <child>             < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!              < --- VisitorXX.onChildText
            <child>             < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
    </message>
    VisitorXX.executeExecutionLifecycleCleanup
```

このライフサイクルメソッドでは、**SAXVisitor** 実装の 1 つのフラグメント実行の周囲にスコープされたリソースを関連する **ExecutionContext** のためにクリーンアップできます。

バグの報告

14.47. EXECUTIONCONTEXT

ExecutionContext は状態情報を保存するためのコンテキストオブジェクトです。**Smooks.filterSource** メソッドの 1 回の実行に特化して、スコープされています。すべての Smooks ビジター実装は、1 つの **Smooks.filterSource** 実行のコンテキスト内でステートレスである必要があり、ビジター実装を **Smooks.filterSource** メソッドの複数の同時実行で使用できるようにします。**ExecutionContext** インスタンスに保存されているすべてのデータは、**Smooks.filterSource** 実行の完了時、失われます。**ExecutionContext** は、すべてのビジター API メッセージイベント呼び出しで提供されます。

バグの報告

14.48. APPLICATIONCONTEXT

ApplicationContext は状態情報を保存するためのコンテキストオブジェクトです。関連する **Smooks** インスタンスの周囲にスコープされます。つまり、1つの **Smooks** インスタンスにつき、1つの **ApplicationContext** インスタンスのみが存在します。このコンテキストオブジェクトは、複数の **Smooks.filterSource** 実行で維持およびアクセスする必要があるデータを保存するために、使用できます。コンポーネント (**SAXVisitor** コンポーネントを含む) は、**ApplicationContext** クラスプロパティを宣言し、**@AppContext** アノテーションを付けて、関連する **ApplicationContext** インスタンスにアクセスできます。以下の例を参照してください。

```
public class MySmooksComponent
{
    @AppContext
    private ApplicationContext appContext;

    // etc...
}
```

バグの報告

第15章 APACHE CAMEL との統合

15.1. APACHE CAMEL-SMOOKS との統合

Camel-Smooks との統合により、Apache Camel 内から Smooks のすべての機能にアクセスできます。既存の Smooks 設定を取得し、Camel ルートで使用できます。

バグの報告

15.2. APACHE CAMEL で SMOOKS を使用する方法

Apache Camel で Smooks を使用するには、次の 3 つの方法があります。

- SmooksComponent
- SmooksDataformat
- SmooksProcessor

バグの報告

15.3. SMOOKSCOMPONENT

SmooksComponent は、Smooks を使用して、Camel メッセージ本文を処理する場合に使用できる Camel モジュールです。

バグの報告

15.4. SMOOKSCOMPONENT の使用

1. 次のように、Camel ルート設定にルートを追加します。

```
from("file://inputDir?noop=true")
.to("smooks://smooks-config.xml")
.to("jms:queue:order")
```

2. smooks-config-xml ファイルの値を編集します。ルート設定を見ただけでは、SmooksComponent が生成している出力のタイプを判断することはできません。代わりに、これは exports 要素によって Smooks 設定で表現されます。

注記

Smooks をプログラムで設定する場合は、SmooksProcessor を使用すると、設定できます。

バグの報告

15.5. SMOOKSCOMPONENT の設定

Apache コンポーネントは、Smooks 設定ファイルの後に指定された任意のオプションを使用できます。現在、SmooksComponent に使用できるオプションは 1 つだけです。

- reportPath: これは、生成される Smooks 実行レポートへのパス (ファイル名を含む) です。

バグの報告

15.6. SMOOKSDATAFORMAT

SmooksDataFormat は、Apache Camel *DataFormat* です。あるデータ形式を別のデータ形式に変換し、元に戻すことができます。これは、あるフォーマットから別のフォーマットへの変換のみを行い、他の Smooks 機能を使用する必要がない場合に使用します。

バグの報告

15.7. SMOOKSDATAFORMAT の例

このコード例は、SmooksDataFormat を使用して、コンマ区切りの値文字列を Customer オブジェクトインスタンスの java.util.List に変換する方法を示しています。

```
SmooksDataFormat sdf = new SmooksDataFormat("csv-smooks-unmarshal-config.xml");
from("direct:unmarshal")
.unmarshal(sdf)
.convertBodyTo(List.class)
.to("mock:result");
```

バグの報告

15.8. SMOOKSPROCESSOR

SmooksProcessor を使用すると、Smooks を完全に制御できます。(たとえば、基礎となる Smooks インスタンスをプログラムで作成する場合は、このコンポーネントを使用します。) SmooksProcessor を使用している場合は、Smooks をプログラムで設定し、Smooks インスタンスをそのコンストラクターに渡すことができます。

バグの報告

15.9. SMOOKSPROCESSOR の例

この例は、Apache Camel ルートで SmooksProcessor を使用する方法を示しています。

```
Smooks smooks = new Smooks("edi-to-xml-smooks-config.xml");
ExecutionContext context = smooks.createExecutionContext();
...
SmooksProcessor processor = new SmooksProcessor(smooks, context);
```

```
from("file://input?noop=true")
.process(processor)
.to("mock:result");
```

Similar to the SmooksComponent we have not specified the result type that Smooks produces (if any that is). Instead this is expressed in the Smooks configuration using the exports element or you can do the same programmatically like this:

```
Smooks smooks = new Smooks();
ExecutionContext context = smooks.createExecutionContext();
smooks.setExports(new Exports(StringResult.class));
SmooksProcessor processor = new SmooksProcessor(smooks, context);
...
from("file://input?noop=true")
.process(processor)
.to("mock:result");
```

バグの報告

15.10. CAMEL プロパティー

表15.1 Camel プロパティー

名前	説明
camel-dataformat	この例は、SmooksDataFormat の使用方法を示しています。(DataFormat は、Camel の org.apache.camel.spi.DataFormat を実装するクラスです。)
camel-integration	この例は、Camel SmooksComponent の使用方法を示しています。(smooks://file:./configs/smooks-config.xml)
splitting-camel	この例は、Smooks と Apache Camel を使用して、UN/EDIFACT メッセージ交換を処理し、個別の交換メッセージを Java および XML フラグメントに分割し、Apache Camel を使用して、フラグメントをルーティングする方法を示しています。

バグの報告

第16章 SMOOKS と JBOSS ENTERPRISE SOA PLATFORM の統合

16.1. MESSAGE TRANSFORMATION

JBoss Enterprise SOA Platform のメッセージ変換機能は、SmooksAction コンポーネントによって提供されます。これは、Smooks フレームワークを ESB アクション処理パイプラインにプラグインできるようにする ESB アクションモジュールです。これにより、Smooks を公開し、設定ファイルを必要とする ESB アクションをセットアップできます。

注記

JBoss Enterprise SOA Platform 製品には、多くの変換クイックスタートのサンプルが付属しています。**transform_** クイックスタートを確認して、それらがどのように動作するかを理解してください。XML2XML の例は、学習に適しています。

バグの報告

16.2. RESOURCE-CONFIG プロパティー

最も基本的な設定では、メッセージ変換は、Smooks 設定ファイルを参照する resource-config プロパティーを使用します。

resource-config プロパティーの値には、**URIResourceLocator** クラスで定義される任意の URI ベースのリソースを指定できます。

```
<action name="transform" class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
        <property name="resource-config" value="/smooks-config.xml" ></property>
</action>
```

バグの報告

16.3. 入出力設定

このアクションは、**MessagePayloadProxy** を使用して、メッセージペイロードにアクセスします。デフォルトでは、Message.Body.DEFAULT_LOCATION 位置からペイロードを取得するように設定されています。また、処理後は、ここに戻します。

注記

これらのデフォルト設定をオーバーライドするには、get-payload-location および set-payload-location アクションプロパティを変更します。

バグの報告

16.4. JAVA 出力設定

1. メッセージを Java オブジェクトに変換するには、**Transform_XML2POJO** クイックスタートの例を確認してください。

2. 構成された Java オブジェクトモデルをモデルドリブンの変換の一部として使用します。パイプラインで SmooksAction の後に続く他の ESB アクションインスタンスで、それらを使用できるようにすることもできます。このような Java オブジェクトグラフは、このアクションによって出力された ESB メッセージに添付され、後続のアクションに入力されるため、後続のパイプラインアクションインスタンスで使用できます。

3. Smooks Java Bean 設定に定義されているように、beanId オブジェクトに直接基づくキーの下で、**Message.getBody().add(...)** を使用して、オブジェクトを ESB メッセージ出力に添付します。つまり、オブジェクトは、**Body.get(beanId)** 呼び出しを実行すると、ESB メッセージ本文で利用できます。

4. また、java-output-location プロパティーを追加すると、出力メッセージに完全な Java オブジェクトマップを添付できます。

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
        <property name="resource-config" value="/smooks-config.xml" ></property>
        <property name="java-output-location" value="order-message-objects-map" >
</property>
    </action>
```

5. これはマップを Default Message Body Location にバインドするための簡単な方法です。

```
 <action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
        <property name="resource-config" value="/smooks-config.xml" ></property>
        <property name="java-output-location" value="$default" ></property>
    </action>
```

バグの報告

16.5. プロファイルベースの変換

1. プロファイルベースの変換を実行するには、次の例を参照してください。1 つの変換設定ファイル (**smooks-config.xml**) を定義する必要があります。メッセージは、3 つの異なるソースと 1 つのターゲット宛先の間で交換されています。ESB は、宛先サービスに送信する前に、3 つの異なるソースのそれぞれから (異なる形式で) 提供されたメッセージを Java オブジェクトに変換する必要があります。ESB の観点から見ると、宛先には、1 つのサービス設定があります。

```
<service category="ServiceCat" name="TargetService" description="Target Service">
        <listeners>
            <!-- Message listners for getting the message into the action pipeline... -->
            <jms-listener name="Gateway-Listener" busidref="quickstartGwChannel" is-
gateway="true"></jms-listener>
            <jms-listener name="Service-Listener" busidref="quickstartEsbChannel"></jms-
listener>
        </listeners>
        <actions>
```

```
            <action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
                <property name="resource-config" value="/smooks-config.xml"></property>
            </action>

            <!-- An action to process the Java Object(s) -->
            <action name="process" class="com.acme.JavaProcessor" ></action>

        </actions>
    </service>
```

2. ソースごとに 1 つずつ、3 つの異なる変換を定義します。これは、*Smooks Message Profiling* を使用すると、実行されます。

3. 定義を 3 つの Smooks 設定ファイル (**from_source1.xml**、**from_source2.xml**、および **from_source3.xml**) に保存します。

4. 上記の各ファイルで、その設定セットに default-target-profile を指定します。

```
 <smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd" default-target-
profile="from:source1">
        <!-- Source1 to Target Java message transformation resource configurations... -->
    </smooks-resource-list>
```

5. これらのファイルをトップレベルの **smooks-config.xml** に追加します。

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.0.xsd">
        <import file="classpath:/from_source1.xml" ></import>
        <import file="classpath:/from_source2.xml" ></import>
        <import file="classpath:/from_source3.xml" ></import>
    </smooks-resource-list>
```

これで、3 つの異なる変換を持つ 1 つの SmooksAction インスタンスをロードするように、システムが設定されました。それぞれが独自の一意のプロファイル名で定義されています。

6. SmooksAction が与えられたメッセージに 3 つの変換のうちどれを適用するかを知るには、メッセージが SmooksAction に流入する前に、メッセージの from プロパティを設定する必要があります。これは、ソース自体で行うことも、SmooksAction の前にあるコンテンツベースのアクションで行うこともできます。

注記

JBoss Enterprise SOA Platform は、from、つまり、from-type、to、および to-type 以外に、他のプロファイルもサポートします。これらを組み合わせて使用すると、より複雑な交換ベースの変換を行うことができます。

バグの報告

16.6. TRANSFORM_XML2POJO2

メッセージ変換は、**/samples/quickstarts/transform_XML2POJO2/** と呼ばれるクイックスタートとして実装できます。このクイックスタートには、2 つのメッセージソースがあります。クイックスタートは、受信メッセージの from プロファイルを検出および設定するアクションパイプライン上で、**Groovy**

スクリプトを実行します。

バグの報告

16.7. TRANSFORM_XML2POJO2 ファイル

変換で使用されるファイルは次のとおりです。

- **jboss-esb.xml**: これは JBoss ESB 設定ファイルです。
- **smooks-config.xml**: このファイルには、トップレベルの変換設定が含まれます。
- **from-dvdstore.xml**: これには、トップレベルの **smooks-config.xml** ファイルにインポートされる DVD ストアメッセージ変換設定が含まれます。(プロファイル設定をメモします。) これは、DVD ストアメッセージを Java オブジェクトに変換するように設計された設定です。
- **from-petstore.xml**: このファイルには、トップレベルの **smooks-config.xml** ファイルにインポートされる Pet Store メッセージ変換設定が含まれます。(プロファイル設定をメモします。) これは、DVD ストアメッセージを Java オブジェクトに変換するように設計された設定です。
- **check-origin.groovy**: これは、内容に基づいて、各メッセージの送信元を判別する単純な **Groovy** スクリプトです。

バグの報告

付録A 更新履歴

改訂 5.3.1-79.400 2013-10-31 **Rüdiger Landmann**
Rebuild with publican 4.0.0

改訂 5.3.1-79 Wed Feb 06 2013 **David Le Sage**
Built from Content Specification: 9506, Revision: 371839 by dlesage