Оглавление

Отзывы **14**

Листинги **18**

Предисловие **29**

Как пользоваться книгой 30
Примеры исходного кода 32
Благодарности 33
От издательства 34

Глава 1. Введение 35

1.1. Кратчайшая история конкурентности 35
1.2. Преимущества потоков 37
1.2.1. Задействование множества процессоров 37
1.2.2. Простота моделирования 38
1.2.3. Упрощенная обработка асинхронных событий 39
1.2.4. Более отзывчивые пользовательские интерфейсы 39
1.3. Риски для потоков 40
1.3.1. Угрозы безопасности 40
1.3.2. Сбои жизнеспособности 43
1.3.3. Угрозы производительности 44
1.4. Потоки есть везде 44

ЧАСТЬ I. ОСНОВЫ **47**

Глава 2. Потокобезопасность 48

2.1. Что такое потокобезопасность? 51
2.1.1. Пример: сервлет без поддержки внутреннего состояния 52
2.2. Атомарность 53
2.2.1. Состояния гонки 54
2.2.2. Пример: состояния гонки в ленивой инициализации 55
2.2.3. Составные действия 57

2.3. Блокировка 58
2.3.1. Внутренние замки́ 60
2.3.2. Повторная входимость 62
2.4. Защита состояния с помощью замков 63
2.5. Живучесть и производительность 66

Глава 3. Совместное использование объектов 70
3.1. Видимость 70
3.1.1. Устаревшие данные 72
3.1.2. Неатомарные 64-разрядные операции 73
3.1.3. Блокировка и видимость 74
3.1.4. Волатильные переменные 75
3.2. Публикация и ускользание 77
3.2.1. Приемы безопасного конструирования 80
3.3. Ограничение одним потоком 81
3.3.1. Узкоспециальное ограничение одним потоком 82
3.3.2. Ограничение стеком 82
3.3.3. ThreadLocal 83
3.4. Немутируемость 85
3.4.1. Финальные поля 87
3.4.2. Пример: использование volatile для публикации немутируемых объектов 88
3.5. Безопасная публикация 90
3.5.1. Ненадлежащая публикация: хорошие объекты становятся плохими 90
3.5.2. Немутируемые объекты и безопасность при инициализации 91
3.5.3. Приемы безопасной публикации 92
3.5.4. Фактически немутируемые объекты 93
3.5.5. Мутируемые объекты 94
3.5.6. Безопасное совместное использование объектов 95

Глава 4. Компоновка объектов 96
4.1. Проектирование потокобезопасного класса 96
4.1.1. Сбор требований к синхронизации 97
4.1.2. Операции, зависимые от состояния 98
4.1.3. Владение состоянием 99
4.2. Ограничение одним экземпляром 100
4.2.1. Мониторный шаблон Java 102
4.2.2. Пример: трекинг такси 103

4.3. Делегирование потокобезопасности 105
4.3.1. Пример: трекер такси с использованием делегирования 107
4.3.2. Независимые переменные состояния 109
4.3.3. Случаи безуспешного делегирования 110
4.3.4. Публикация базовых переменных состояния 111
4.3.5. Пример: трекер такси, публикующий свое состояние 112
4.4. Добавление функциональности в существующие потокобезопасные классы 114
4.4.1. Блокировка на стороне клиента 115
4.4.2. Компоновка 117
4.5. Документирование политик синхронизации 118
4.5.1. Толкование расплывчатой документации 119

Глава 5. Строительные блоки 121
5.1. Синхронизированные коллекции 121
5.1.1. Проблемы синхронизированных коллекций 121
5.1.2. Итераторы и исключение ConcurrentModificationException 124
5.1.3. Скрытые итераторы 125
5.2. Конкурентные коллекции 127
5.2.1. ConcurrentHashMap 128
5.2.2. Дополнительные атомарные операции над ассоциативным массивом 129
5.2.3. CopyOnWriteArrayList 129
5.3. Блокирующие очереди и паттерн «производитель-потребитель» 130
5.3.1. Пример: поиск на рабочем столе 132
5.3.2. Серийное ограничение одним потоком 133
5.3.3. Двухсторонние очереди и кража работы 135
5.4. Блокирующие и прерываемые методы 136
5.5. Синхронизаторы 137
5.5.1. Защелки 137
5.5.2. FutureTask 138
5.5.3. Семафоры 141
5.5.4. Барьеры 143
5.6. Создание эффективного масштабируемого кэша результатов 145
Итоги 153

ЧАСТЬ II. СТРУКТУРИРОВАНИЕ КОНКУРЕНТНЫХ ПРИЛОЖЕНИЙ 155

Глава 6. Выполнение задач 156
6.1. Выполнение задач в потоках 156
6.1.1. Последовательное выполнение задач 157
6.1.2. Явное создание потоков для задач 158
6.1.3. Недостатки создания неограниченных потоков 159
6.2. Фреймворк Executor 160
6.2.1. Пример: веб-сервер с использованием Executor 161
6.2.2. Политики выполнения 162
6.2.3. Пулы потоков 163
6.2.4. Жизненный цикл исполнителя Executor 164
6.2.5. Отложенные и периодические задачи 166
6.3. Поиск эксплуатационно-пригодного параллелизма 167
6.3.1. Пример: последовательный страничный отрисовщик 168
6.3.2. Задачи, приносящие результаты: Callable и Future 169
6.3.3. Пример: страничный отрисовщик с объектом Future 171
6.3.4. Ограничения параллелизации разнородных задач 172
6.3.5. CompletionService: исполнитель Executor встречается с очередью BlockingQueue 174
6.3.6. Пример: страничный отрисовщик со службой CompletionService 175
6.3.7. Наложение временны́х ограничений на задачи 176
6.3.8. Пример: портал бронирования поездок 177
Итоги 179

Глава 7. Отмена и выключение 180
7.1. Отмена задачи 181
7.1.1. Прерывание 183
7.1.2. Политики прерывания 186
7.1.3. Отклик на прерывание 188
7.1.4. Пример: хронометрированный прогон 190
7.1.5. Отмена с помощью Future 191
7.1.6. Работа с непрерываемым блокированием 193
7.1.7. Инкапсуляция нестандартной отмены с помощью newTaskFor 194
7.2. Остановка поточной службы 196
7.2.1. Пример: служба журналирования 197
7.2.2. Выключение службы ExecutorService 201
7.2.3. Ядовитые таблетки 202

7.2.4. Пример: служба однократного выполнения....203
7.2.5. Ограничения метода shutdownNow....204
7.3. Обработка аномальной терминации потоков....207
7.3.1. Обработчики неотловленных исключений....208
7.4. Выключение JVM....210
7.4.1. Хуки....210
7.4.2. Потоки-демоны....211
7.4.3. Финализаторы....212
Итоги....213

Глава 8. Применение пулов потоков....214
8.1. Неявные стыковки между задачами и политиками выполнения....214
8.1.1. Взаимная блокировка с ресурсным голоданием....215
8.1.2. Длительные задачи....217
8.2. Определение размера пула потоков....217
8.3. Конфигурирование класса ThreadPoolExecutor....219
8.3.1. Создание и удаление потоков....219
8.3.2. Управление задачами очереди....221
8.3.3. Политика насыщения....223
8.3.4. Фабрики потоков....224
8.3.5. Настройка класса ThreadPoolExecutor после конструирования....227
8.4. Расширение класса ThreadPoolExecutor....228
8.4.1. Пример: добавление статистики в пул потоков....229
8.5. Параллелизация рекурсивных алгоритмов....230
8.5.1. Пример: фреймворк головоломки....232
Итоги....238

Глава 9. Приложения с GUI....239
9.1. Почему GUI-интерфейсы являются однопоточными?....239
9.1.1. Последовательная обработка событий....241
9.1.2. Ограничение одним потоком в Swing....241
9.2. Кратковременные задачи GUI....242
9.3. Длительные задачи GUI....245
9.3.1. Отмена....247
9.3.2. Индикация хода выполнения и завершения....248
9.3.3. SwingWorker....249
9.4. Совместные модели данных....249
9.4.1. Потокобезопасные модели данных....252
9.4.2. Раздвоенные модели данных....252
9.5. Другие формы однопоточных подсистем....253
Итоги....254

ЧАСТЬ III. ЖИЗНЕСПОСОБНОСТЬ, ПРОИЗВОДИТЕЛЬНОСТЬ И ТЕСТИРОВАНИЕ 255

Глава 10. Предотвращение сбоев жизнеспособности 256

10.1. Взаимная блокировка 256

10.1.1. Взаимные блокировки из-за порядка блокировки 258

10.1.2. Взаимная блокировка из-за динамического порядка следования замков 260

10.1.3. Взаимные блокировки между взаимодействующими объектами 264

10.1.4. Открытые вызовы 266

10.1.5. Ресурсные взаимные блокировки 269

10.2. Предотвращение и диагностирование взаимной блокировки 270

10.2.1. Хронометрированные замки 270

10.2.2. Анализ взаимной блокировки с помощью поточных дампов 271

10.3. Другие сбои жизнеспособности 274

10.3.1. Голодание 274

10.3.2. Слабая отзывчивость 275

10.3.3. Активная блокировка 276

Итоги 277

Глава 11. Производительность и масштабирование 278

11.1. Некоторые мысли о производительности 279

11.1.1. Производительность и масштабируемость 280

11.1.2. Оценивание компромиссов производительности 282

11.2. Закон Амдала 284

11.2.1. Пример: сериализация, скрытая в фреймворках 287

11.2.2. Качественное применение закона Амдала 289

11.3. Стоимость, вносимая потоком 290

11.3.1. Переключение контекста 290

11.3.2. Синхронизации памяти 291

11.3.3. Блокирование 294

11.4. Сокращение конфликта блокировки 295

11.4.1. Сужение области действия замка («вошел, вышел») 296

11.4.2. Сокращение степени детализации замка 298

11.4.3. Чередование блокировок 301

11.4.4. Недопущение горячих полей 302

11.4.5. Альтернативы исключающим блокировкам 305

11.4.6. Мониторинг задействованности процессоров 305

11.4.7. Объединению объектов в пул — «нет»! 307

11.5. Пример: сравнение производительности ассоциативного массива Map 309
11.6. Сокращение издержек на переключение контекста 311
Итоги 313

Глава 12. Тестирование конкурентных программ 315
12.1. Тестирование на правильность 317
12.1.1. Базовые модульные тесты 319
12.1.2. Тестирование блокирующих операций 320
12.1.3. Тестирование на безопасность 322
12.1.4. Тестирование на управление ресурсами 328
12.1.5. Использование обратных вызовов 330
12.1.6. Генерирование большего числа расслоений 331
12.2. Тестирование на производительность 332
12.2.1. Расширение теста PutTakeTest за счет хронометрирования 333
12.2.2. Сравнение многочисленных алгоритмов 337
12.2.3. Измерение отзывчивости 338
12.3. Предотвращение ошибок при тестировании 340
12.3.1. Сбор мусора 341
12.3.2. Динамическая компиляция 341
12.3.3. Нереалистичный отбор ветвей кода 343
12.3.4. Нереалистичные уровни конфликта 344
12.3.5. Устранение мертвого кода 345
12.4. Комплементарные подходы к тестированию 347
12.4.1. Ревизия кода 348
12.4.2. Инструменты статического анализа 348
12.4.3. Аспектно-ориентированные методы тестирования 351
12.4.4. Средства профилирования и мониторинга 351
Итоги 352

ЧАСТЬ IV. ПРОДВИНУТЫЕ ТЕМЫ 353

Глава 13. Явные замки 354
13.1. Lock и ReentrantLock 354
13.1.1. Опрашиваемое и хронометрируемое приобретение замка 356
13.1.2. Прерываемое приобретение замка 359
13.1.3. Неблочно структурированная замковая защита 360
13.2. Соображения по поводу производительности 360
13.3. Справедливость 362

13.4. Выбор между synchronized и ReentrantLock ... 365
13.5. Замки чтения-записи ... 366
Итоги ... 371

Глава 14. Построение настраиваемых синхронизаторов ... 372
14.1. Управление зависимостью от состояния ... 373
14.1.1. Пример: распространение сбоя предусловия на вызывающие элементы кода ... 374
14.1.2. Пример: грубая блокировка с помощью опрашивания и сна ... 377
14.1.3. Очереди условий на освобождение ... 379
14.2. Использование очередей условий ... 382
14.2.1. Условный предикат ... 382
14.2.2. Слишком раннее пробуждение ... 384
14.2.3. Пропущенные сигналы ... 386
14.2.4. Уведомление ... 386
14.2.5. Пример: шлюзовый класс ... 389
14.2.6. Вопросы безопасности подклассов ... 390
14.2.7. Инкапсулирование очередей условий ... 392
14.2.8. Протоколы входа и выхода ... 393
14.3. Явные объекты условий ... 393
14.4. Анатомия синхронизатора ... 397
14.5. AbstractQueuedSynchronizer ... 399
14.5.1. Простая защелка ... 401
14.6. AQS в классах синхронизатора библиотеки java.util.concurrent ... 403
14.6.1. ReentrantLock ... 403
14.6.2. Semaphore и CountDownLatch ... 405
14.6.3. FutureTask ... 406
14.6.4. ReentrantReadWriteLock ... 407
Итоги ... 407

Глава 15. Атомарные переменные и неблокирующая синхронизация ... 409
15.1. Недостатки замковой защиты ... 410
15.2. Аппаратная поддержка конкурентности ... 412
15.2.1. Сравнить и обменять ... 413
15.2.2. Неблокирующий счетчик ... 415
15.2.3. Поддержка операции CAS в JVM ... 417
15.3. Классы атомарных переменных ... 417
15.3.1. Атомарные компоненты в качестве «более качественных волатильных» ... 419
15.3.2. Сравнение производительности: замки против атомарных переменных ... 420

15.4. Неблокирующие алгоритмы 424
15.4.1. Неблокирующий стек 425
15.4.2. Неблокирующий связный список 427
15.4.3. Обновители атомарных полей 431
15.4.4. Проблема ABA 433
Итоги 434

Глава 16. Модель памяти Java 435
16.1. Что такое модель памяти и зачем она нужна? 435
16.1.1. Платформенные модели памяти 437
16.1.2. Переупорядочивание 438
16.1.3. Модель памяти Java в менее чем 500 словах 440
16.1.4. Совмещение за счет синхронизации 443
16.2. Публикация 445
16.2.1. Небезопасная публикация 446
16.2.2. Безопасная публикация 447
16.2.3. Идиомы безопасной инициализации 448
16.2.4. Блокировка с двойной проверкой 450
16.3. Безопасность инициализации 452
Итоги 454

ПРИЛОЖЕНИЕ А. АННОТАЦИИ ДЛЯ КОНКУРЕНТНОСТИ 456
A.1. Аннотации классов 456
A.2. Аннотации полей и методов 457

Библиография 459

2 Потокобезопасность

Возможно, вас удивит, что конкурентное программирование связано с потоками или замка́ми[1] не более, чем гражданское строительство связано с заклепками и двутавровыми балками. Разумеется, строительство мостов требует правильного использования большого количества заклепок и двутавровых балок, и то же самое касается построения конкурентных программ, которое требует правильного использования потоков и замко́в. Но это всего лишь *механизмы* — средства достижения цели. Написание потокобезопасного кода — это, по сути, управление доступом к *состоянию* и, в частности, к *совместному* (shared) *мутируемому состоянию* (mutable state).

В целом *состояние* объекта — это его данные, хранящиеся в *переменных состояния* (state variables), таких как экземплярные и статические поля или поля из других зависимых объектов. Состояние хеш-массива `HashMap` частично хранится в самом объекте `HashMap`, но также и во многих объектах `Map.Entry`. Состояние объекта включает любые данные, которые могут повлиять на его поведение.

---

[1] В тексте встречаются термины lock и block, которые часто переводятся одним словом «блокировка», что может подразумевать и объект, и процесс. В английском языке для процесса блокирования как приостановки продвижения есть термин blocking. Под термином lock имеется в виду «замок», «замковый защитный механизм». Во избежание путаницы термин lock переводится, как замок, кроме устоявшихся выражений, где принят перевод «блокировка». Замок — это механизм контроля доступа к данным с целью их защиты. В программировании замки часто используются для того, чтобы несколько программ или программных потоков могли использовать ресурс совместно, например, обращаться к файлу для его обновления на поочередной основе. — *Примеч. науч. ред.*

К *совместной* переменной могут обратиться несколько потоков, *мутируемая* — меняет свое значение. На самом деле мы пытаемся защитить от неконтролируемого конкурентного доступа не код, а *данные*.

Создание потокобезопасного объекта требует синхронизации для координации доступа к мутируемому состоянию, невыполнение которой может привести к повреждению данных и другим нежелательным последствиям.

Всякий раз, когда более чем один поток обращается к переменной состояния и один из потоков, возможно, в нее пишет, все потоки должны координировать свой доступ к ней с помощью синхронизации. Синхронизацию в Java обеспечивают ключевое слово `synchronized`, дающее эксклюзивную *блокировку*, а также волатильные (`volatile`) и атомарные переменные и явные замки́.

Удержитесь от соблазна думать, что существуют ситуации, не требующие синхронизации. Программа может работать и проходить свои тесты, но оставаться неисправной и завершиться аварийно в любой момент.

Если многочисленные потоки обращаются к одной и той же переменной, имеющей мутируемое состояние, без соответствующей синхронизации, то *ваша программа неисправна*. Существует три способа ее исправить:

- *не использовать* переменную состояния совместно во всех потоках;
- сделать переменную состояния *немутируемой*;
- при каждом доступе к переменной состояния использовать *синхронизацию*.

Исправления могут потребовать значительных проектных изменений, поэтому *гораздо проще проектировать класс потокобезопасным сразу, чем модернизировать его позже*.

Будут или нет многочисленные потоки обращаться к той или иной переменной, узнать сложно. К счастью, объектно-ориентированные технические решения, которые помогают создавать хорошо организованные и удобные в сопровождении классы — такие как инкапсуляция и сокрытие данных, — также помогают создавать потокобезопасные классы. Чем меньше потоков имеет доступ к определенной переменной, тем проще обеспечить синхронизацию и задать условия, при которых к данной переменной можно обращаться. Язык Java не заставляет вас инкапсулировать состояние — вполне допустимо хранить состояние в публичных

полях (даже публичных статических полях) или публиковать ссылку на объект, который в иных случаях является внутренним, — но чем лучше инкапсулировано состояние вашей программы, тем проще сделать вашу программу потокобезопасной и помочь сопроводителям поддерживать ее в таком виде.

> При проектировании потокобезопасных классов хорошие объектно-ориентированные технические решения: инкапсуляция, немутируемость и четкая спецификация инвариантов — будут вашими помощниками.

Если хорошие объектно-ориентированные проектные технические решения расходятся с потребностями разработчика, стоит поступиться правилами хорошего проектирования ради производительности либо обратной совместимости с устаревшим кодом. Иногда абстракция и инкапсуляция расходятся с производительностью — хотя и не так часто, как считают многие разработчики, — но образцовая практика состоит в том, чтобы сначала делать код правильным, а *затем* — быстрым. Старайтесь задействовать оптимизацию только в том случае, если измерения производительности и потребности говорят о том, что вы обязаны это сделать[1].

Если вы решите, что вам необходимо нарушить инкапсуляцию, то не все потеряно. Вашу программу по-прежнему можно сделать потокобезопасной, но процесс будет сложнее и дороже, а результат — ненадежнее. Глава 4 характеризует условия, при которых можно безопасно смягчать инкапсуляцию переменных состояния.

До сих пор мы использовали термины «потокобезопасный класс» и «потокобезопасная программа» почти взаимозаменяемо. Строится ли потокобезопасная программа полностью из потокобезопасных классов? Не обязательно: программа, которая состоит полностью из потокобезопасных классов, может не быть потокобезопасной, и потокобезопасная программа может содержать классы, которые не являются потокобезопасными. Вопросы, связанные с компоновкой потокобезопасных классов, также

[1] В конкурентном коде следует придерживаться этой практики даже больше, чем обычно. Поскольку ошибки конкурентности чрезвычайно трудно воспроизводимы и не просты в отладке, преимущество небольшого прироста производительности на некоторых редко используемых ветвях кода может вполне оказаться ничтожным по сравнению с риском, что программа завершится аварийно в условиях эксплуатации.

рассматриваются в главе 4. В любом случае понятие потокобезопасного класса имеет смысл только в том случае, если класс инкапсулирует собственное состояние. Термин «потокобезопасность» может применяться к *коду*, но он говорит о *состоянии* и может применяться только к тому массиву кода, который инкапсулирует его состояние (это может быть объект или вся программа целиком).

2.1. Что такое потокобезопасность?

Дать определение потокобезопасности непросто. Быстрый поиск в Google выдает многочисленные варианты, подобные этим:

> ...может вызываться из многочисленных потоков программы без нежелательных взаимодействий между потоками.

> ...может вызываться двумя или более потоками одновременно, не требуя никаких других действий с вызывающей стороны.

Учитывая подобные определения, неудивительно, что мы находим потокобезопасность запутанной! Как отличить потокобезопасный класс от небезопасного? Что мы вообще подразумеваем под словом «безопасный»?

В основе любого разумного определения потокобезопасности лежит понятие *правильности* (correctness).

Правильность подразумевает *соответствие* класса *своей спецификации*. Спецификация определяет *инварианты* (invariants), ограничивающие состояние объекта, и *постусловия* (postconditions), описывающие эффекты от операций. Как узнать, что спецификации для классов являются правильными? Никак, но это не мешает нам их использовать после того, как мы убедили себя, что код работает. Поэтому давайте допустим, что однопоточная правильность — это нечто видимое. Теперь можно предположить, что потокобезопасный класс ведет себя правильно во время доступа из многочисленных потоков.

> Класс является *потокобезопасным*, если он ведет себя правильно во время доступа из многочисленных потоков, независимо от того, как выполнение этих потоков планируется или перемежается рабочей средой, и без дополнительной синхронизации или другой координации со стороны вызывающего кода.

Многопоточная программа не может быть потокобезопасной, если она не является правильной даже в однопоточной среде[1]. Если объект реализован правильно, то никакая последовательность операций — обращения к публичным методам и чтение или запись в публичные поля — не должна нарушать его инварианты или постусловия. *Ни один набор операций, выполняемых последовательно либо конкурентно на экземплярах потокобезопасного класса, не может побудить экземпляр находиться в недопустимом состоянии.*

Потокобезопасные классы инкапсулируют любую необходимую синхронизацию сами и не нуждаются в помощи клиента.

2.1.1. Пример: сервлет без поддержки внутреннего состояния

В главе 1 мы перечислили структуры, которые создают потоки и вызывают из них компоненты, за потокобезопасность которых ответственны вы. Теперь мы намерены разработать сервлетную службу разложения на множители и постепенно расширить ее функционал, сохраняя потокобезопасность.

В листинге 2.1 показан простой сервлет, который распаковывает число из запроса, раскладывает его на множители и упаковывает результаты в отклик.

Листинг 2.1. Сервлет без поддержки внутреннего состояния

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

Класс `StatelessFactorizer`, как и большинство сервлетов, не имеет внутреннего состояния: не содержит полей и не ссылается на поля из других классов. Состояние для конкретного вычисления существует только в локальных

[1] Если нестрогое использование термина *правильность* здесь вас беспокоит, то вы можете думать о потокобезопасном классе как о классе, который неисправен в конкурентной среде, как и в однопоточной среде.

переменных, которые хранятся в потоковом стеке и доступны только для выполняющего потока. Один поток, обращающийся к `StatelessFactorizer`, не может повлиять на результат другого потока, делающего то же самое, поскольку эти потоки не используют состояние совместно.

> Объекты без поддержки внутреннего состояния всегда являются потокобезопасными.

Тот факт, что большинство сервлетов могут быть реализованы без поддержки внутреннего состояния, значительно снижает бремя по обеспечению потокобезопасности самих сервлетов. И только когда сервлеты должны что-то запомнить, требования к их потокобезопасности возрастают.

2.2. Атомарность

Что происходит при добавлении элемента состояния в объект без поддержки внутреннего состояния? Предположим, мы хотим добавить счетчик посещений, который измеряет число обработанных запросов. Можно добавить в сервлет поле с типом `long` и приращивать его при каждом запросе, как показано в `UnsafeCountingFactorizer` в листинге 2.2.

Листинг 2.2. Сервлет, подсчитывающий запросы без необходимой синхронизации. *Так делать не следует*

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```