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);
    }
}
```