Многопоточное программирование и Java

Корректность, паттерны, подходы

Евгений Кирпичёв

18 апреля 2010 г.

Обсуждение:
http://community.livejournal.com/ru_java/929773.html

Цель доклада

Расширить кругозор в области многопоточного программирования вообще и на Java в частности.

- Корректность
  - Как сформулировать, повысить, гарантировать
- Инструменты
- Паттерны и велосипеды
- Подходы и фишки разных языков

Классификация многопоточных программ

Parallelism vs Concurrency

Parallel Одна задача бьется на много частей

Concurrent Много разных задач (*сложнее*)

Сегодня — в основном о concurrency.

Проблемы многопоточного программирования

Писать concurrent программы — трудно.

- Обычно слабая поддержка со стороны языка
- Трудно не ошибиться
- Очень трудно тестировать

Классификация многопоточных программ (contd.)

Программы бывают разные.

	Raytracer	Server	IDE
Сколько задач?	Много	Много	Мало
Зависимы?	Нет	Нет	Да
Общие ресурсы?	Нет	Да	Да
Трудность	CPU perf	IO perf	Корректность

Часть 1. Корректность.

Содержание

Корректность

- Причины багов
- Доказательство корректности
- Улучшение корректности

Паттерны и подходы

- Параллелизм
- Concurrency

Cutting edge

- Haskell
- Erlang
- .NET
- Clojure

Заключение

Appendix

- Java memory model
- Производительность
- Тестирование и наблюдаемость

Причины багов

Корень всех зол — изменяемое во времени состояние.

- N нитей, K состояний $\Rightarrow O(K^N)$ переплетений
- Промежуточные состояния *видимы*
- Даже вызов метода — это больше не черный ящик
  - Его тело больше не атомарно
- Корректные многопоточные программы не модульны
  - Sequentialy correct + Sequentially correct $\neq$ Concurrently correct

Причины багов

Sequentialy correct $+$ Sequentially correct $\neq$ Concurrently correct

Доказуемая корректность

Face it:

- Либо программа *доказуемо* корректна
- Либо она, почти наверняка, некорректна
  - Пусть вероятность бага 10^{-8}
  - 10^9 вызовов за неделю — и вероятность $1 - e^{-10} \approx 1$
  - При непрерывном тестировании ошибка тоже всплывет только через неделю
- Работает *на бумаге* или не работает вообще

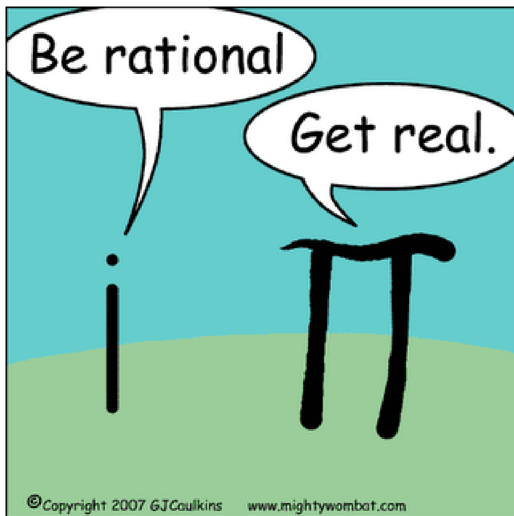
Доказуемая корректность

Face it:

- Либо программа *доказуемо* корректна
- Либо она, почти наверняка, некорректна

Tony Hoare:

There are two ways of constructing a software design.
One way is to make it so simple that there are ***obviously no deficiencies****, and the other way is to make it so complicated that there are* ***no obvious deficiencies****.*
The first method is far more difficult.



Формальные методы

Применять не обязательно.

Формальные методы

А зачем тогда они вообще нужны?

- Чтобы описывать систему *полуформально*
- Чтобы чуять: формализуема ли система *в принципе*?
  - Если нет (система слишком запутана) — о корректности можно забыть.

Формальные методы

Мой опыт: достаточно *попытаться* формализовать систему:

- Всплывают противоречия, недоговорки в требованиях
- Выделяются компоненты без четкой спецификации
  - Блажен, кто верует, что они „и так“ работают

Формальные методы

Что же это за методы?

Например, линейная темпоральная логика.

Доказуемая корректность

Линейная темпоральная логика — способ рассуждать о последовательностях событий.

Программа моделируется автоматом („структурой Крипке“).

Исследуются возможные последовательности состояний.

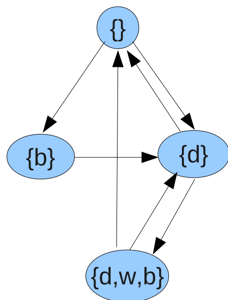
Структура Крипке

Структура Крипке $\approx$ граф состояний.

- Множество состояний
- Некоторые *начальные*
- Некоторые соединены *переходом*
  - Переход *безусловный*
- Множество *фактов*
- В каждом состоянии верны некоторые факты.

Структура Крипке

Микроволновка.



d=”дверь закрыта”
w=”включено излучение”
b=”зажата кнопка Пуск”

Структура Крипке

Переходы не подписаны, потому что:

- Номер состояния *полностью* описывает состояние программы
- Если переход условный — надо поделить состояние на два
  - То, в котором условие верно (переход *может* произойти)
  - То, в котором условие неверно (переход *не может* произойти)
- Программу можно странслировать в структуру Крипке
- Структуры Крипке прекрасно поддаются верификации

LTL

Линейная темпоральная логика (Linear Temporal Logic).
Рассуждает о последовательностях состояний мира во времени.

- Есть запрос $\Rightarrow$ когда-нибудь будет ответ
- Блокировка запрошена $\Rightarrow$ клиент будет обслужен сколь угодно большое число раз, пока не отпустит ее
- ...

Прелести:

- Краткая, понятная, мощная и однозначная
- LTL-свойства легко верифицируются даже вручную
    - Относительно структур Крипке
- Много мощных инструментов

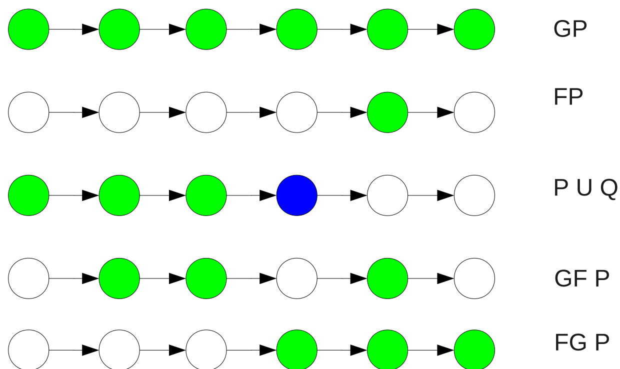
Формулы LTL

$p_1, p_2, \ldots$	Переменные
$A \wedge B, \neg A \ldots$	Логические связки
XA	В следующий момент A (neXt)
$GA \equiv \Box A$	Всегда A (Globally)
$FA \equiv \Diamond A$	Когда-нибудь A (Future)
$A \mathcal{U} B$	A, по крайней мере до наступления B

Бесконечности

- Начиная с некоторого момента, навсегда: $FG\ A \equiv \Diamond\Box A$
- Бесконечно часто: $GF\ A \equiv \Box\Diamond A$

Иллюстрация формул LTL



Свойства

- Живость (liveness)
  - Что-то хорошее когда-нибудь произойдет: $\Diamond Alive$
  - Задание когда-нибудь начнется
- Безопасность (safety)
  - Что-то плохое никогда не произойдет: $\Box \neg Dead$
  - Задание никогда не потерпит крах

Свойства

Справедливость (fairness)

- То, что может произойти — произойдет
- Слабая: $\Diamond\Box Req \to \Box\Diamond Ack$
  - Ресурс запрошен (и остается запрошенным) $\to$ когда-нибудь будет выдан
- Сильная: $\Box\Diamond Req \to \Box\Diamond Ack$
  - Нить бесконечно много раз *исполнима* (имеет не самый низкий приоритет из неблокированных) $\to$ когда-нибудь получит квант
- Антоним — starvation (голодание)

Примеры LTL-свойств

Устройство не выключается само по себе

$\Box(isOn \rightarrow \quad (isOn\ \mathcal{U}\ (turnOffPressed \vee noBattery))$

Не реже чем раз в 5 тактов включается контроллер двигателя

$\Box F^5(activeThread == ENGINE_CONTROLLER)$

Model checking

Наука о проверке свойств программ (например, LTL) на основе их моделей (например, структур Крипке) — *Model checking*.

Премия Тьюринга 2007 — за достижения в области model checking, сделавшие его применимым для проектов реального размера.

Инструмент: SPIN

Повсеместно используемый язык + верификатор LTL-свойств для многозадачных систем.
Лауреат ACM Software System award (2001) (другие лауреаты: Unix, TeX, Java, Apache, TCP/IP, ...).

- Си-подобный синтаксис
- Для нарушенных свойств генерируется нарушающая их трасса
- (уродливый, но терпимый) GUI
- Я проверял: Пригодно для применения на практике.

`http://spinroot.com/spin/whatispin.html` — официальный сайт.

AspectJ+LTL

http://www.sable.mcgill.ca/~ebodde/rv/JLO/ — J-LO: проверяет LTL-свойства в рантайме (заброшен).

http://abc.comlab.ox.ac.uk/extensions — Tracecheck: продолжение, от тех же авторов: проверяет регулярные выражения над трассами программы.

Model checking:further reading

http://www.inf.unibz.it/~artale/FM/slide3.pdf
Туториал по LTL.

http:
//citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3062
Статья про J-LO

http://spinroot.com/spin/Doc/course/mc-tutorial.pdf
Model Checking: A tutorial overview (много ссылок)

Карпов Ю. Г. — Model checking (продается в магазинах).

http://savenkov.lvk.cs.msu.su/mc.html
Интересный курс в МГУ (WIP).

Ближе к делу

Как рассуждать о реальных программах?

- Аксиоматика: Java memory model и т.п.
- Основные параметры (документируйте их!):
  - Thread safety
  - Возможность блокирования; используемые блокировки и порядок их взятия
  - Верхняя граница времени выполнения (константа или нет)
  - Реакция на прерывание
  - Реакция на ошибки
  - Реакция на таймауты
  - ...

Java Memory Model

С ней необходимо ознакомиться. Много сюрпризов.

- Нельзя создавать нити в конструкторе
- Не-final поля не потокобезопасны (даже если не меняются)
- ...

См. appendix, книгу JCIP, JLS.

Thread safety

- Immutable. У объекта всего 1 состояние.
- Thread-safe. Каждая операция сама по себе безопасна/атомарна. Последовательности — нет.
  - Последовательности нужно координировать извне
  - Документируйте, какие *именно* все-таки безопасны
  - Пример: Банковский счет

Thread safety

- Thread-compatible. Можно использовать многопоточно, если не вызывать операции одновременно.
  - Нужна внешняя синхронизация, но с ней будет работать
- Thread-hostile. Вообще нельзя использовать многопоточно.
- Thread-confined. Можно использовать из *конкретной* нити.
  - Пример: Swing

CheckThread

Набор аннотаций для документирования thread safety.
Есть плагин к IDEA.
Очень полезно.

Пример

```
@ThreadSafe class SynchronizedInteger {
  @GuardedBy("this") private int value;
  synchronized int get()       { return value; }
  synchronized void set(int v) { value = v;    }
}
```

Java Path Finder

Виртуальная машина, умеющая символьно выполнять Java-код и верифицировать свойства *для всех возможных запусков с учетом многопоточности*. Сделано в NASA.
`http://javapathfinder.sourceforge.net/`
`http://sourceforge.net/projects/visualjpf/` — GUI
`http://www.visserhome.com/willem/presentations/ase06jpftut.ppt` — подобие туториала

- Утверждается, что справляется с проектами до 10KLOC.
- Находил баги в NASA-овском софте (в марсоходе, например)
- Основан на SPIN
- Есть ant-task и плагин к eclipse; расширяем

Я написал маленький пример с дедлоком. Сработало.

Другие тулы

Статический анализ многопоточных ошибок есть и в:

- PMD
- FindBugs (есть плагин к IDEA)

Проверено: помогает (и не только от многопоточных).

Философское избавление от багов

Надо уменьшить $O(K^N)$.

- Уменьшить число *наблюдаемых* состояний (K)
  - Immutability
  - Инкапсуляция состояния
  - Высокоуровневые операции
  - Факторизация
  - Синхронизация
- Навязать трассам эквивалентность
  - Идемпотентные операции
    - `foo();foo();` $\sim$ `foo();`
  - Коммутативные (независимые) операции
    - `foo();bar();` $\sim$ `bar();foo();`
    - В т.ч. инкапсуляция и независимые объекты
- Считать большее число трасс корректными (ослабить требования)

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (Java Memory Model)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* . . .

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (Java Memory Model)
  - Иначе изменение может произойти "тайком"
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (Java Memory Model)
  - Иначе изменение может произойти "тайком"
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (Java Memory Model)
  - Иначе изменение может произойти "тайком"
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (Java Memory Model)
  - Иначе изменение может произойти "тайком"
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Инкапсуляция состояния

Программа корректна ровно настолько, насколько корректны и скоординированы все, кто *могут* повлиять на состояние объекта.

Мораль: Чем меньше способов повлиять на состояние, тем лучше.

- Мало кому давать к нему доступ
  - Мало кого надо будет координировать
- Давать доступ в терминах малого числа высокоуровневых операций
  - Мало комбинаций операций надо рассматривать
  - Меньше нарушается целостность между операциями

Высокоуровневые операции

Определение: *Высокоуровневая операция — операция, не нарушающая целостность системы.*

- `AtomicInteger.getAndIncrement`
- `ConcurrentMap.putIfAbsent`
- `Bank.transfer`

Достаточно правильно реализовать саму операцию.
Проектируйте API в таком стиле, если это возможно.

Высокоуровневые операции

Частный случай:

- У объекта в меру большое состояние
- Хочется атомарно поменять несколько его частей
- Запихните состояние в объект и поменяйте атомарно ссылку на этот объект
  - `AtomicReference`
- *Чисто функциональные структуры данных* всегда потокобезопасны

http://tobega.blogspot.com/2008/03/java-thread-safe-state-design-pattern.html

См.тж. Okasaki, Purely functional data structures.

Факторизация

Программа корректна ровно настолько, насколько корректны и скоординированы все, кто *могут* повлиять на состояние объекта.

Мораль: Независимые аспекты состояния выносить в независимые объекты.

Сложность падает с $(K_1 + K_2)^N$ до $K_1^N + K_2^N$ (бред, но суть примерно такая).

Факторизация

```
class DataLoader {
    ClientToken beginLoad(Client client);
    void writeData(ClientToken token, Data data);
    void commit(ClientToken token);
    void rollback(ClientToken token);
}
```

Факторизация

Сложный код:

- Таблица `ClientToken`/транзакция
- (многопоточные клиенты) `synchronized(locks.get(token))`

Факторизация

Разорвем лишнюю зависимость между независимыми задачами.

```
class DataLoader {
    PerClientLoader beginLoad(Client client);
}
class PerClientLoader {
    void writeData(Data data);
    void commit();
    void rollback();
}
```

Факторизация

- Простой и незагрязненный код
- Синхронизация простым `synchronized`
- Никаких таблиц
- Никаких проблем с охраной таблиц от многопоточного доступа

Синхронизация (сериализация доступа)

Это все знают: “Хочешь защитить объект — сделай на нем `synchronized`”.

11455

Синхронизация (сериализация доступа)

Если надо защитить доступ к комбинации объектов — надо синхронизироваться *в одинаковом порядке*.
Иначе — дедлоки.

Так делать не надо

```
void foo() {
  synchronized(x) {
    synchronized(y) {
      ...
    }
  }
}
void bar() {
  synchronized(y) {
    synchronized(x) {
      ...
    }
  }
}
```

Борьба с дедлоками

Нужен глобально фиксированный порядок блокировок.
“Экзотический” случай:

Так делать не надо

```
void transfer(Account from,Account to,int money) {
  synchronized(from) {
    synchronized(to) {
      ..
    }
  }
}
```

Борьба с дедлоками: Экзотический случай

Нужен глобально фиксированный порядок блокировок.

Надо делать вот так

```
void transfer(Account from,Account to,int money) {
  synchronized(from<to ? from : to) {
    synchronized(from?to ? to : from) {
      ..
    }
  }
}
```

Подробно — см. Java Concurrency in Practice.

Еще немного о дедлоках

Не стоит вызывать чужие методы (callbacks), держа блокировку (“alien call”). Такие вызовы лучше превратить в “open call” (вызов вне блокировки)

```
private void frobbleBar(Bar bar) {
    List<BarListener> copy;
    synchronized (this) {
        copy = new ArrayList<BarListener>(listeners);
    }
    for (Listener l : snapshot)
        l.barFrobbled(bar);
}
```

Более общий случай:

- Минимизируйте сложность и “неизвестность” кода внутри блокировки
- Совсем хорошо, если его длительность *ограничена сверху*

Еще немного о дедлоках

Частая причина дедлоков — ресурсы и пулы.

- Доступ к ресурсам надо упорядочивать, как и блокировки
- Пулы + взаимозависимые задачи = проблема Thread starvation deadlock
  - Пул забит нитями, ждущими результата вычисления
  - Вычисление в очереди, но его некому начать — пул забит

Синхронизация и наследование

Очень трудно сделать эффективный thread-safe класс на основе non-thread-safe базового.

- Придется перегрузить *все до единой* операции
- В т.ч. и длительные, тяжеловесные
- Не дай бог какие-то из операций `final`

Как с наследованием:

- “Design for inheritance or prohibit it”
- “Design for thread-safety”

Синхронизация и наследование

Паттерн “Thread-safe interface”:

- Поделите методы компонента на 2 части
  - final интерфейс + реализация
- Синхронизация только в „интерфейсе“
- „Реализация“ вызывает *только* „реализацию“
- Наследники не беспокоятся о синхронизации
- Довольно общий паттерн
  - синхронизация $\rightarrow$ транзакция БД, транзакция undo, ...

Идемпотентные операции

`foo();foo();` $\sim$ `foo();`

Тогда можно не заботиться об ограничении числа вызовов `foo()` (но `foo()` должна быть thread-safe).

Особенно важно в распределенных и веб-системах (с ненадежным подтверждением вызова `foo()`).

Частый пример — ленивая инициализация.

Идемпотентные операции

- `set.add(x)` — идемпотентна
- `x.setFoo(foo)` — идемпотентна
- `list.add(x)` — не идемпотентна
- `/buy.php?product_id=123` — не идемпотентна

Идемпотентные операции

Пример достижения идемпотентности:

- Присвоить каждому вызову операции уникальный `id`
- Если такой `id` уже был — это дубль
- Актуально в вебе.

Ослабление требований

Иногда обеспечить жесткие инварианты целостности невозможно.

- Они могут нарушаться извне
  - Есть внешние процессы, нарушающие целостность
  - Они нам не подконтрольны
  - Например: синхронизация файлов, изменяемых внешними процессами
- Их слишком дорого поддерживать
  - Например, большая распределенная система
  - Синхронное проталкивание изменения на все реплики — слишком долго
  - С учетом падений узлов все еще хуже
- Лень писать сложный код :)

Отрицательная обратная связь

Судя по всему, один из мощнейших приемов проектирования некоторых классов многозадачных систем.
Суть:

- „Состояние системы всегда целостно“ — очень трудно и не всегда нужно.
- „Состояние системы все целостнее и целостнее“ — легко обеспечить, часто достаточно.

Делаем систему лишь *стремящейся* к состоянию целостности.

Отрицательная обратная связь

Пример структуры:

- Система периодически просыпается и осматривается
- Вычисляет, чем мир отличается от „идеала“
  - Какие-то данные с чем-то не согласованы
  - Какие-то готовые к обработке данные еще не обрабатываются
  - Какие-то данные обрабатываются уже слишком долго; наверное, обработчик умер
  - ...
- Выполняет действия.

Отрицательная обратная связь

Плюсы подхода „система как сходящийся процесс“:

- **Отрицательная обратная связь**
  - Последствия большинства проблем рано или поздно исчезнут.
- **Высокая модульность**
  - Система распадается на набор чисто линейных автоматов-„обработчиков“.
- **Очень простая отладка и тестирование**
  - Если что-то не так — система просто приняла неправильное решение в одном из состояний.
  - Может, само рассосется, можно не фиксить

Отрицательная обратная связь

- **Высокая устойчивость**
  - Система неуязвима для „смертельных“ багов (corrupted синглтоны, утечки ресурсов).
  - Сделать состояние persistent, перезапускать процесс.
- **Готовность к распределенной работе**
  - Сделать состояние глобально видимым и обеспечить простейшую блокировку.

Отрицательная обратная связь

Применимость:

- Dataflow-like задачи (данные текут по системе с рядом этапов обработки)
- Репликация и синхронизация
- Арбитраж ресурсов и load balancing
- ...

Eventual consistency

Относится к распространению данных и изменений по распределенной системе.
„Каждое изменение когда-нибудь станет видно всей системе“.
Видимо, единственная эффективная модель в действительно больших системах (Amazon, eBay и т.п.)

CAP Theorem

Choose any 2:
Consistency: все узлы видят систему одинаково
Availability: запрос к живому узлу обязательно получает ответ
Partition tolerance: любое сообщение между узлами может быть утеряно

Eventual consistency

Consistency, Availability, Partition Tolerance.

ACID[1] → BASE[2]

ACID

Atomicity

Consistency

Isolation

Durability

BASE

Basically **A**vailable

Soft state

Eventually consistent

Soft state

The state ... could be lost in a crash without *permanent* disruption of the service features being used.

Clark, “The design philosophy of the DARPA internet protocols”

[1] Кислота

[2] Щелочь

Further reading

- http://www.webperformancematters.com/journal/2007/8/21/asynchronous-architectures-4.html — о масштабировании в Amazon. Очень полезные советы.
- http://www.infoq.com/interviews/dan-pritchett-ebay-architecture — о масштабировании в eBay
- http://roc.cs.berkeley.edu/ — Исследовательский проект “Recovery-Oriented Computing”
  - Бороться с последствиями ошибок, поскольку искоренить все причины невозможно
  - Несколько основных принципов (изоляция, избыточность, undo, самодиагностика, перезапускаемость)
  - Очень разумно, если подумать.

Абстрагирование

Многопоточную программу намного легче написать и отладить, если в ней нет ничего, кроме самой сути.
Баги вылезают на поверхность, нет соблазна сказать “В нашей задаче такая ситуация невозможна”.
Алгоритмы легче отлаживать по отдельности.

- Не “блокировка индекса на чтение или запись”, а просто “блокировка на чтение или запись”
- Не “очередь e-mail'ов”, а “отказоустойчивая очередь задач”
- Не “параллельный подсчет слов”, а MapReduce
- Не “параллельная фильтрация массива”, а ParallelArray

Может оказаться, что такая штука уже есть.

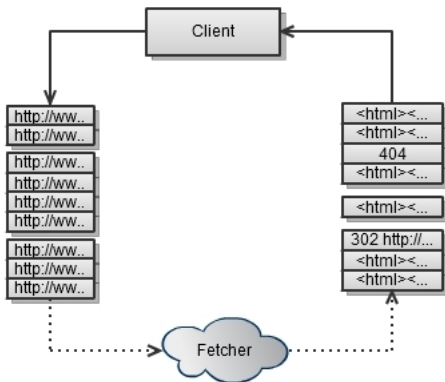
Абстрагирование

А какие штуки уже есть?
Об этом — позже.

Абстрагирование: пример

Привязка к внешней системе скачивания урлов.

- Задания пишутся в файлы
- Внешняя система их подхватывает
- Ответы тоже пишутся в файлы
- Надо предоставить асинхронный API
  - `void loadPage(url, onOk, onError)`


Client
http://ww..
http://ww..
http://ww..
http://ww..
http://ww..
http://ww..
http://ww..
http://ww..
http://ww..
<html><...
<html><...
404
<html><...
<html><...
302 http://...
<html><...
<html><...
Fetcher

Абстрагирование: пример

Devil in the details:

- Не зафлудить систему (блокироваться, если много заданий „в полете“)
- Следить за таймаутами (долго не отвечает → `onError`)
- Не допускать гонок между ответами и таймаутами
- Беречь нитку слежения за ответами (вызывать `onOk`,`onError` в отдельных нитях)
- Обрабатывать редиректы
- Организовать синхронный shutdown

Абстрагирование: пример

Было: Нетестируемая каша, делающая все сразу.

Абстрагирование: пример

Стало:

- Семафор для ограничения числа заданий „в полете“
- Абстрактный паттерн „Stoppable“
- Абстрактный „батчер“
- Абстрактный „поллер“
- Абстрактный „таймаутер“
- Всё элементарно тестируется (нашлись баги)
- Всё полезно и повторно используемо

Абстрагирование: пример

```
interface Stoppable {
  void stopSynchronously() throws InterruptedException;
  void stopEventually();
}
```

Абстрагирование: пример

```
class Batcher<T> {
  Batcher(int periodMs);
  Stoppable start();
  abstract void flush(T[] ts);
  void submit(T t);
}
```

Абстрагирование: пример

```
class Poller<T> {
  Poller(int periodMs);
  Stoppable start();
  abstract @Nullable T poll();
  abstract void process(@NotNull T t);
}
```

Абстрагирование: пример

```
class Timeouter<T> {
  Poller(int periodMs);
  Stoppable start();
  void submit(T t, int timeoutMs);
  abstract void onTimedOut(T t);
}
```

Абстрагирование: пример

Итог: Из бизнес-логики полностью пропал многопоточный код.

Часть 2. Паттерны и подходы.

Содержание

Корректность
Причины багов
Доказательство корректности
Улучшение корректности
Паттерны и подходы
Параллелизм
Concurrency
Cutting edge
Haskell
Erlang
.NET
Clojure
Заключение
Appendix
Java memory model
Производительность
Тестирование и наблюдаемость

Модели многопоточных вычислений

Параллелизм:

- MapReduce: Огромные объемы данных, распределенная обработка
- Fork/Join: Задачи с рекурсивной структурой
- Векторные модели: huge-scale SIMD
- Конвейер / data flow
- ...

Concurrency:

- Message-passing (actor model)
- Event loop (message-passing с одним / несколькими одинаковыми акторами)
  - Сервера, большинство GUI-библиотек, в т.ч. Swing
- Software Transactional Memory
- Асинхронное программирование

MapReduce

Слишком известно, чтобы на нем останавливаться

Реализация для Java: Hadoop

Fork/Join

```
if(isSmall(problem)) {
    return solveSequentially(problem);
} else {
    // Fork
    List<Problem> subproblems = split(problem);
    List<Result> res = parallelMap(solve, subproblems);
    // Join
    return combine(res);
}
```

Fork/Join

Пример: Шахматы. Предпросмотр на n шагов.

- `n==0` $\Rightarrow$ оценить позицию
- Fork: По подзадаче на каждый ход
- Join: Выбрать ход с самой высокой оценкой

Fork/Join

Пример: Интегрирование.

- Интервал мал $\Rightarrow$ проинтегрировать влоб
- Fork: по подзадаче на обе половины интервала
- Join: сложить результаты

Fork/Join

Пример: Умножение матриц: $C+ = AB$ (В т.ч. поиск путей в графах и т.п.)

- A и B малы $\Rightarrow$ влоб
- Fork: Поделить A на верх/низ и B на лево/право
- Join: Ничего, просто дождаться результатов Fork

Fork/Join

A1
A2

B1 B2

11	12
21	22

Cij += Ai Bj

Fork/Join in JDK7

Примерно так. Официального окончательного Javadoc до сих пор нет.

```
class Solver extends RecursiveAction {
    private final Problem problem;
    int result;
    protected void compute() {
        if (problem.size < THRESHOLD) {
            result = problem.solveSequentially();
        } else {
            int m = problem.size / 2;
            Solver left, right;
            left = new Solver(problem.subProblem(0, m));
            right = new Solver(problem.subProblem(m, problem.size));
            forkJoin(left, right);
            result = //get the result from left right
        }
    }
}
ForkJoinExecutor pool = new ForkJoinPool(
  Runtime.availableProcessors());
Solver solver = new Solver(new Problem());
pool.invoke(solver);
```

Векторные алгоритмы

Huge-scale SIMD: Примитивные операции оперируют сразу над большими массивами.

- Map: Применение функции к каждому элементу / склейка N массивов по N-арной операции
- Fold: Ассоциативная свертка в моноиде.

$$\star \text{ ассоциативна: } \quad a \star (b \star c) = (a \star b) \star c$$
$$u \text{ — единица } \star\text{: } \quad a \star u = u \star a = a$$

$$fold \ \star \ u \ [x_0, x_1, \ldots, x_n] = u \star x_0 \star x_1 \star \ldots \star x_n$$

- Scan (Prefix sum): Пробег (Префиксная сумма)

$$scan \ \star \ u \ [x_0, x_1, \ldots, x_n] = [u, u \star x_0, u \star x_0 \star x_1, \ldots, u \star \ldots \star x_n]$$

Векторные модели

Прелести:

- Свертка параллелизуется: Ускорение $O(P/log\ P)$ на P процессорах (в $log\ P$ раз хуже линейного)
- Пробег так же параллелизуется
- Удивительно мощные и универсальные операции

Префиксные суммы

Что можно сделать сверткой/пробегом (т.е. параллельно):

- sum,min,max,avg,...
- Radix sort, Quicksort
- Сложение длинных чисел
- Выпуклая оболочка
- Многие алгоритмы на графах
- Многие рекуррентные последовательности n-го порядка
- Трехдиагональные системы уравнений
- ...

Префиксные суммы

Особенно интересно, *как* это делается. Базовые операции поверх пробега:

- Рекуррентная последовательность 1го порядка
- Упаковка массива
- Сортировка массива по 1-битному флагу (“разбиение”)
- Сегментированный пробег (пробег со сбросами)

Префиксные суммы

Упаковка массива (“pack”, “filter”).

1	2	3	4	5	6	7	8	9

2	3	5	7	8	9

Префиксные суммы

Сортировка по 1-битному флагу (“разбиение”, “partition”).

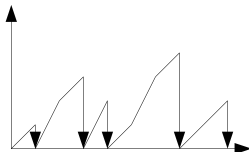
1	2	3	4	5	6	7	8	9

1	4	6	2	3	5	7	8	9

Radixsort тривиален.

Префиксные суммы

Сегментированный пробег (Segmented scan).



Позволяет реализовать рекурсию без рекурсии.

Префиксные суммы

В целом — красивый, мощный, простой в реализации и необычный векторный “язык”.
Часто используют на GPU.
`http://sourceforge.net/projects/amino-cbbs/` — реализация на Java (содержит много других готовых параллельных алгоритмов).
`http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf` — статья Prefix sums and their applications: обзор техник и применений.
`http://www.cs.cmu.edu/~blelloch/papers/Ble90.pdf` — Vector models for data-parallel computing, целая книга про то же. Строжайше рекомендую.

ParallelArray

New in JDK7. Реализация некоторых векторных примитивов поверх Fork/Join framework.

- Стандартные: Map, Zip, Filter, Fold (`reduce`), Scan (`cumulate` + `precumulate`), Sort
- Поинтереснее: Сечение по диапазону (`withBounds`), Перестановка (`replaceWithMappedIndex`)
- Функциональный до мозга костей API (подавляющее большинство функций — ФВП)
  - Правда, без фанатичного следования функциональной чистоте
- Ленивые операции
- Сплавка операций (*fusion*)

ParallelArray

Примерчик.

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

public class Student {
    String name;
    int graduationYear;
    double gpa;
}

static final Ops.Predicate<Student> isSenior = new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};
static final Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
};
```

Message-passing concurrency

Фактически единственный подход в распределенных системах.

- Вообще нет разделяемого состояния
- *Акторы* обмениваются *сообщениями*
- У каждого актора есть mailbox (FIFO-очередь сообщений)
- Отсылка сообщений асинхронная и неблокирующая

Message-passing concurrency

- Не панацея: дедлоки есть (но другие)
- Тем не менее, написать корректную программу — проще
- Есть формальные модели (CSP) $\Rightarrow$ поддается верификации

Message-passing concurrency

Реализации:

- Куча языков: MPI
- Termite Scheme
- Clojure: agents
  - Очень интересная и мощная вариация. Definitely worth seeing. `http://clojure.org/agents`
  - "Clojure is to concurrent programming as Java was to OOP"
- Scala: actors
- Java: Kilim
- Haskell: CHP (Communicating Haskell Processes)
- И, конечно, Erlang
  - Killer feature: Selective receive

Message-passing concurrency

Kilim:

- Легковесные нити (1млн в порядке вещей, сверхбыстрое переключение)
- Очень быстрый message passing (утверждается, что 3x Erlang)
- Постпроцессит байт-код (CPS transform для методов, аннотированных `@pausable`).

Message-passing concurrency

Further reading:
http://kilim.malhar.net/
http://www.cl.cam.ac.uk/research/srg/opera/publications/papers/kilim_ecoop08.pdf — научная статья. Не для робких, но читать можно. Много интересных идей.

http://eprints.kfupm.edu.sa/30514/1/30514.pdf — оригинальная книга Тони Хоара про Communicating Sequential Processes — теоретическая основа message-passing

Event loop

Вариация на тему message passing, но — не много взаимодействующих акторов, а один (или несколько одинаковых) жирный и всемогущий.

- Серверы (и `accept`, и `select/poll/...` сюда относятся)
- GUI-фреймворки

Прелести:

- Это не фреймворк, но паттерн: легко реализовать и использовать даже на микроуровне
- К нему сводится куча задач
- Очень легко пишется
- Большинство многопоточных проблем отсутствуют

Event loop

```
private BlockingQueue<OurEvent> events =
  new LinkedBlockingQueue<OurEvent>();

void mainLoop() {
    while(true) {
        processEvent(events.poll());
    }
}

public class OurEvent {...}
```

Software Transactional Memory

Транзакции на уровне памяти; атомарные куски кода.

- Полноценный commit/rollback, полная атомарность
- Транзакции *повторяемы*
  - retry при коммите, если извне изменена прочтенная переменная
- Транзакционные программы комбинируемы!
  - В отличие от shared state (правильно+правильно≠правильно)
- Не должно быть *никаких* побочных эффектов, кроме обращений к транзакционной памяти

Software Transactional Memory

Псевдокод:

```
// Insert a node into a doubly-linked list atomically
atomic {
    newNode->prev = node;
    newNode->next = node->next;
    node->next->prev = newNode;
    node->next = newNode;
}
```

Software Transactional Memory

Псевдокод (впервые предложено в реализации в Haskell):

```
atomic {
    if (queueSize > 0) {
        remove item from queue and use it
    } else {
        retry
    }
}
```

`retry` перезапускает транзакцию в момент, когда изменится одна из *прочитанных* ею переменных.

Software Transactional Memory

Реализации:

- Haskell: полноценная и безопасная поддержка
- Clojure: полноценная и небезопасная поддержка
- Java: Deuce, JSTM, ...
- Много библиотек для C и C++, в т.ч. даже компиляторы с ключевым словом `atomic`
  - Не знаю, используют ли их

Software Transactional Memory

Further reading:
`http://research.microsoft.com/Users/simonpj/papers/stm/stm.pdf` — Composable Memory Transactions.

`http://themonadreader.files.wordpress.com/2010/01/issue15.pdf` — Журнал Monad.Reader Issue 15, содержит статью “Implementing STM in pure Haskell”, интересное обсуждение вопросов реализации STM.

Асинхронное программирование

Набирает обороты благодаря вебу (AJAX) и большим распределенным системам, где синхронный RPC слишком дорог.
Quite a challenge.

Асинхронное программирование

Синхронный API:

```
interface API {
   Answer compute(Question request) throws Whoops;
}
```

Асинхронный API:

```
interface API {
   void compute(Question request,
                {Answer => void} onReady,
                {Whoops => void} onWhoops)
}
```

Асинхронное программирование

Причины:

- Блокирующие вызовы неудобны или не поддерживаются
  - AJAX
  - GUI („асинхронно получить результат диалога“ и т.п.)
- Асинхронный API допускает более эффективную реализацию
  - Асинхронный ввод-вывод
  - Асинхронный доступ к хранилищам данных ...
  - Активная сторона (сервер, сетевой драйвер ОС ...) может решать, когда и в каком порядке отвечать на запросы

Асинхронное программирование

Проблемы:

- Нелинейный поток исполнения
- Очень неудобно отлаживать
  - Пошаговое выполнение в отладчике не работает
- Нет аналогов привычных конструкций управления (циклы, `try..finally` ...)
- Даже сэмулировать их нелегко
- В большинстве ЯП асинхронный код крайне громоздок

Асинхронное программирование

```
public void requestInit() {
    AdminConsoleService.App.getInstance().getLoadGeneratorServersInfo(
      new AcAsyncCallback<List<ServerInfo>>() {
        public void doOnSuccess(final List<ServerInfo> loadGenerators) {
            AdminConsoleService.App.getInstance().getApplicationServersInfo(
              new AcAsyncCallback<List<ServerInfo>>() {
                public void doOnSuccess(List<ServerInfo> appServers) {
                    init(appServers, loadGenerators);
                }
            });
        }
    });
}
```

(c) ru_java.

Асинхронное программирование

Как же быть?

Асинхронное программирование

Как же быть? Вспоминаем уроки функционального программирования.

- Избавиться от синтаксического мусора
  - Асинхронные значения — первоклассные объекты (см. далее)
  - Нужен язык с замыканиями. Sorry guys.
- Реализовать структуры управления
- Не сойти с ума от сложности

Асинхронное программирование

Как же быть? Вспоминаем уроки функционального программирования.

- Избавиться от синтаксического мусора
- Реализовать структуры управления
  - Циклы через рекурсию
  - Эмуляция хвостовых вызовов
  - Все это стоит абстрагировать в „комбина́торную библиотеку“
- Не сойти с ума от сложности

Асинхронное программирование

Как же быть? Вспоминаем уроки функционального программирования.

- Избавиться от синтаксического мусора
- Реализовать структуры управления
- Не сойти с ума от сложности
  - Целый класс проблем, специфичных для этого подхода
  - Большинство связано с mutable state

Асинхронное программирование

Основная идея:

```
interface Async<T> {
  void run(Callback<T> onOk, Callback<Throwable> onError)
}
```

Эти объекты *комбинируемы*.

Асинхронное программирование

Учимся у .NET вообще и F# в частности.

```
let AsyncHttp(url:string) =
   async {  // Create the web request object
            let req = WebRequest.Create(url)

            // Get the response, asynchronously
            let! rsp = req.GetResponseAsync()

            // Grab the response stream and a reader. Clean up when we're done
            use stream = rsp.GetResponseStream()
            use reader = new System.IO.StreamReader(stream)

            // synchronous read-to-end
            return reader.ReadToEnd() }
```

Это называется „asynchronous workflows“.
Workflow — красивый синоним для слова „Монада“ (google://workflow, monad).
http://blogs.msdn.com/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx

Асинхронное программирование

```
class AsyncPrimitives {
  Async<T> succeed(T value) {...}
  Async<T> failwith(Throwable error) {...}
  Async<U> bind(Async<T> at, Func<T,Async<U>> fau) {...}
}
```

Этого достаточно для большинства структур управления[3].
Это называется „монада“.

[3] При поддержке оптимизации хвостовых вызовов

Асинхронное программирование

Монады проникли в мейнстрим: у C# теперь тоже учимся (LINQ).

```
var requests = new[]
{
    WebRequest.Create("http://www.google.com/"),
    WebRequest.Create("http://www.yahoo.com/"),
    WebRequest.Create("http://channel9.msdn.com/")
};
```

http://www.aboutcode.net/2008/01/14/Async+WebRequest+Using+LINQ+Syntax.aspx

Асинхронное программирование

Монады проникли в мейнстрим: у C# теперь тоже учимся (LINQ).

```
var pages = from request in requests
    select
        from response in request.GetResponseAsync()
        let stream = response.GetResponseStream()
        from html in stream.ReadToEndAsync()
        select new { html, response };
```

http://www.aboutcode.net/2008/01/14/Async+WebRequest+Using+LINQ+Syntax.aspx

Асинхронное программирование

Монады проникли в мейнстрим: у C# теперь тоже учимся (LINQ).

```
foreach (var page in pages)
{
    page(d =>
    {
        Console.WriteLine(d.response.ResponseUri.ToString());
        Console.WriteLine(d.html.Substring(0, 40));
        Console.WriteLine();
    });
 }
```

http://www.aboutcode.net/2008/01/14/Async+WebRequest+Using+LINQ+Syntax.aspx

Мораль: Познайте монады — станет легче писать асинхронный код.

Асинхронное программирование

Further reading:
Очень интересный подход с использованием `yield return`: `http://tomasp.net/blog/csharp-async.aspx`.
Wes Dyer — объяснение монады Cont на C#: `http://blogs.msdn.com/wesdyer/archive/2008/01/11/the-marvels-of-monads.aspx`
Что такое комбинаторная библиотека: `http://fprog.ru/2009/issue3`, статья „Элементы функциональных языков"
Асинхронные комбинаторы на Java: `http://spbhug.folding-maps.org/wiki/EugeneKirpichov`, **слайды** про Java и FP.

Further reading

Разные паттерны многопоточного и распределенного программирования. Очень много и интересно.

- http://www.cs.wustl.edu/~schmidt/patterns-ace.html
- http://parlab.eecs.berkeley.edu/wiki/patterns/patterns — еще круче

Содержание

Корректность
- Причины багов
- Доказательство корректности
- Улучшение корректности

Паттерны и подходы
- Параллелизм
- Concurrency

Cutting edge
- Haskell
- Erlang
- .NET
- Clojure

Заключение

Appendix
- Java memory model
- Производительность
- Тестирование и наблюдаемость

Cutting edge

Что умеет Haskell:

- Все побочные эффекты контролируются
- Нет неконтролируемых проблем из-за их наличия
- Позволяет немыслимые в других языках вещи

Cutting edge

Что умеет Haskell:

- Самые быстрые в мире легковесные нити и примитивы синхронизации
  - 1 место на Language Shootout, 60x быстрее нативных Ubuntu, 5x быстрее Erlang!
- Параллельные стратегии
  - Простые *параллельные* вычисления: нет аналогов в других языках
- Data Parallel Haskell
  - Автоматически распараллеливаемые векторные вычисления
- Транзакционная память
- Работа в направлении GPU
- Мощные библиотеки

Легковесные нити

Nothing special, просто очень быстрые и почти 0 синтаксического оверхеда.
C++0x futures/promises and more в 60 строк.

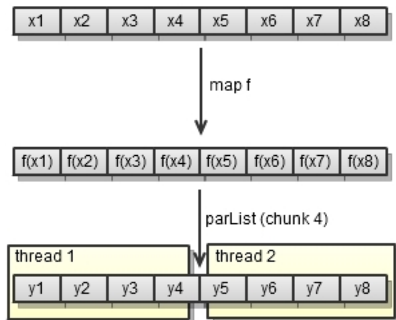
Параллельные стратегии

Ленивость позволяет отделить алгоритм и способ распараллеливания.

Результат алгоритма — структура, полная „обещаний“.

Обещания „исполняются“ (форсируются) с помощью *стратегии*.

Стратегия — это обыкновенная, простая функция.



Идея — ядерная. Это можно реализовать и на Java, если постараться

Параллельные стратегии

Например, стратегия „параллельно форсировать все элементы списка“:

```
parList :: Strategy a -> Strategy [a]
parList strat []      = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Data Parallel Haskell

- Векторизатор (транслирует даже рекурсивные функции в векторные операции)
- Библиотека параллельных векторных операций

```
dotp_double :: [:Double:] -> [:Double:] -> Double
dotp_double xs ys = sumP [:x * y | x <- xs | y <- ys:]

smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul sv v | sv <- sm :]
```

http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/NdpSlides.pdf — слайды

Data Parallel Haskell

```
sort :: [:Float:] -> [:Float:]
sort a = if (length a <= 1) then a
         else sa!0 +++ eq +++ sa!1
 where
    m = a!0
    lt = [: f | f<-a, f<m :]
    eq = [: f | f<-a, f==m :]
    gr = [: f | f<-a, f>m :]
    sa = [: sort a | a <- [:lt,gr:] :]
```

Векторизуется и параллелизуется.

Repa

Regular, shape-polymorphic, parallel arrays in Haskell.
"The Haskell code transparently makes use of multicore hardware"
Сделано группой Мануэля Чакраварти — это один из крутейших имплементоров Хаскелла.
`http://justtesting.org/regular-shape-polymorphic-parallel-arrays-in`

Транзакционная память

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically do
  deposit  to   amount
  withdraw from amount
```

Транзакционная память

```
check True = return ()
check False = retry

limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    check (amount <= 0 || amount <= bal)
    writeTVar acc (bal - amount)
```

Четкое разделение между:

- IO — код с произвольными побочными эффектами
- STM — код, единственный эффект которого — работа с транзакционной памятью
- `atomically :: STM t -> IO t` (наоборот нельзя)

Erlang

- Связь процессов *только* через обмен сообщениями.
- Selective receive (избирательная выборка из mailbox).
  - Мощная фича.
- Прозрачная распределенность.
- Очень эффективная реализация нитей и message passing.
- Очень крутая и продуманная инфраструктура.
- Очень простой язык.
- Активно применяется в индустрии (в т.ч. Amazon, Facebook, Yahoo, ...).

`http://spbhug.folding-maps.org/wiki/Erlang` — самопиар ;)

F#

Asynchronous workflows: уже рассмотрели.

C#

- Asynchronous LINQ (можно сделать самому: LINQ = монады)
  - Уже рассмотрели
- Parallel LINQ

Parallel LINQ (PLINQ)

- `IEnumerable<T>`, `IParallelEnumerable<T>`
- `AsParallel`

http://msdn.microsoft.com/en-us/magazine/cc163329.aspx

Parallel LINQ (PLINQ)

```
IEnumerable<T> data = ...;
var q = from x in data.AsParallel()
        where p(x)
        orderby k(x)
        select f(x);
foreach (var e in q) a(e);
```

Транслируется в комбинацию `ParallelEnumerable.{Select,Where,OrderBy}` и т.п.

Parallel LINQ (PLINQ)

```
IEnumerable<T> leftData = ..., rightData = ...;
var q = from x in leftData.AsParallel()
        join y in rightData on x.a == y.b
        select f(x, y);
```

Clojure

"Clojure is to concurrent programming as Java was to OOP"

- Библиотека чисто функциональных структур данных
- 4 механизма concurrency: Atoms, Vars, Agents, Refs
- **Все они оперируют над immutable значениями и чистыми функциями**
  - Because this is **the** way to go.

Atoms

Более или менее обычные atomic переменные.

`(swap! atom f)` „Атомарно заменить x на $f(x)$, вернуть старый x (compare-and-exchange)“

f обязана быть чистой: ее могут вызвать несколько раз

http://clojure.org/atoms

Vars

Thread-local, dynamically scoped переменные.

`(def x 0)` объявить x с глобальным значением по умолчанию 0.

`x` прочитать значение x.

`(set! x 5)` присвоить $x = 5$ в рамках текущей нити.

`(binding [x 5] (do-something))` выполнить `do-something`, полагая $x = 5$ в рамках текущей нити.

`http://clojure.org/vars`

Agents

Асинхронно изменяемые переменные.

`(agent 0)` агент с начальным значением 0

`(deref x)` текущее значение

`(send x f)` асинхронно заменить x на $f(x)$

`(await x)` дождаться окончания всех действий над агентом

`(set-validator x f)` повесить валидатор состояния

`(add-watch x f)` повесить слушателя

http://clojure.org/agents

Refs

Транзакционная память (STM).

- Довольно обычная
- Транзакции должны быть без побочных эффектов
- Любопытно: `(commute x f)`: заменить x на $f(x)$, предполагая что f коммутирует с остальными.
  - Например, `(commute counter inc)` — подойдет
  - Работает слегка быстрее, чем другие функции-мутаторы

`http://clojure.org/refs`

Содержание

Корректность
- Причины багов
- Доказательство корректности
- Улучшение корректности

Паттерны и подходы
- Параллелизм
- Concurrency

Cutting edge
- Haskell
- Erlang
- .NET
- Clojure

Заключение

Appendix
- Java memory model
- Производительность
- Тестирование и наблюдаемость

Что было

- Классификация многопоточных программ
- Формальные методы (LTL) и тулы (SPIN,JPF,CheckThread)
- Корректность: инкапсуляция, высокоуровневые операции, факторизация, дедлоки, атомарные и идемпотентные операции, отрицательная обратная связь
- Методики: fork/join, векторные алгоритмы, message-passing, event-driven, STM, асинхронное программирование
- Haskell: легкие нити, параллельные стратегии, векторизация, STM
- Erlang: мощная инфраструктура
- Clojure: Atoms, Vars, Agents, Refs
- C#: PLINQ

The end

Спасибо! Вопросы?

Содержание

Корректность
- Причины багов
- Доказательство корректности
- Улучшение корректности

Паттерны и подходы
- Параллелизм
- Concurrency

Cutting edge
- Haskell
- Erlang
- .NET
- Clojure

Заключение

Appendix
- Java memory model
- Производительность
- Тестирование и наблюдаемость

Java Memory Model

Грубо: Набор аксиом о том, какие происшествия в памяти кому и когда видимы.

- Чтобы компилятор мог выполнять оптимизации
  - Переупорядочение, удаление дохлого кода . . .
- Чтобы можно было эффективно использовать кэши, особенно в многопроцессорных системах.

Пример

```
private int x, y;

void foo() {
    x = 5;
    y = 7;
}

void bar() {
    if(x==5) {
        // Неверно, что y==7
    }
}
```

final поля и JMM

Пример

```
public final class Foo {
    private int x;
    Foo(int x) {this.x = x;}
    int getX() {return x;  }
}
```

Не thread-safe! x должен быть final.

Java Memory Model

Более точно: Набор аксиом о том, какие *наблюдения* результатов последовательности действий в памяти — возможны.

Цели:

- Безопасность
  - Безопасная инициализация
- Интуитивность
- Эффективная реализуемость

Java Memory Model

Влияющие факторы:

- `synchronized`
- `volatile`
- `final`

Java Memory Model

Основные понятия:

- Действие
  - Обращение к разделяемым переменным
  - Вход/выход из монитора
  - Запуск/ожидание нитей
- Видимость
- Отношение *happens-before*
  - Жизнь 1 нити упорядочена
  - Жизнь 1 монитора упорядочена
  - Жизнь 1 volatile переменной упорядочена

Что все-таки делает volatile?

Чтение *одной* volatile переменной — почти точка синхронизации.

```
class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42; v = true;
    }

    public void reader() {
        if (v == true) { /* x==42 */ }
    }
}
```

Что все-таки делает final?

После конструирования `final` поля *видимы*.

- Видимы также объекты, доступные через них
- Ссылка на объект не должна утекать из конструктора
  - Регистрация листенером, присваивание статического поля...
  - Делайте это после конструирования.

Производительность

Что тормозит?

- Ждущие нити
- Захват (даже успешный) блокировок
- Переключение контекстов

Производительность

Что делать?

- Не ждать и не блокироваться
- Не переключаться

Уменьшение блокировок

Как не ждать?

- Использовать неблокирующие алгоритмы и операции
- Увеличить гранулярность блокировок
  - Антоним: Один монитор на всю программу
- Использовать алгоритмы, блокирующие не всех
  - ReaderWriterLock

Неблокирующие алгоритмы и операции

Базис:

- `get`, `set`, `getAndSet`
- `boolean compareAndSet(expect, update)` (CAS)
  - Поменять *с expect* на update
  - Если не expect — значит, мы прозевали, как кто-то поменял из другой нити
  - Надо перезапустить кусок алгоритма в надежде, что теперь обойдется без коллизий
- `addAndGet`, `getAndAdd`, `...`

Неблокирующие алгоритмы и операции

Средства в Java:

- `java.util.concurrent.atomic`
- `AtomicInteger`, `AtomicLong`
- `AtomicReference`, `AtomicStampedReference`
- (малоизвестно) `AtomicIntegerArray`, Long, Reference

Атомарные операции

Пример алгоритма на CAS:

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        ...
    }
}
```

http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html

Неблокирующие контейнеры

Готовые неблокирующие структуры данных (New in JDK6):

- ConcurrentSkipListMap
- ConcurrentSkipListSet

Прелести:

- Итераторы никогда не бросают CME

Уменьшение гранулярности блокировок

- Блокируйте не всё
- Блокируйте не всех

Уменьшение гранулярности блокировок

"Блокируйте не всё":
Если надо защитить большую структуру данных, но меняются только ее куски — защищайте куски или группы кусков.
Пример: `ConcurrentHashMap` (new in JDK6). Техника "lock striping".

- Чтение неблокирующее
- Запись блокирующая в $1/k$ случаев, где k настраивается
  - Рекомендуется $k \approx$ число нитей-писателей
- k блокировок защищают k кусков хэш-таблицы
- Итераторы также не бросают CME

Reader/Writer lock

“Блокируйте не всех”:
Чтение совместимо с чтением, но не с записью.
Бывают разные:

- Readers-preference
- Writers-preference
- Arrival-order
- ...

Класс `ReentrantReadWriteLock`. Реализует примерно arrival-order. Если есть ждущий писатель — новые читатели только после него.

Переключение контекстов

Как поменьше переключать контексты?

- Иметь поменьше нитей
  - Например, одну (Event-driven)
- Побольше асихронности

Переключение контекстов

Как поменьше переключать контексты?
Пусть нити не отвлекаются.

- Одни пихают задания в очередь
- Другие вынимают и исполняют их

Цифры

Сколько реально стоят различные операции?	
Атомарные операции и volatile	$\approx$ 0.1мкс; incAndGet 2-3x быстрее CAS
Переключение контекста	Несколько мкс (независимо от числа нитей)
Uncontended synchronized	$\approx$ 0.1-1мкс
Contented synchronized	$\approx$ 20мкс

http://mailinator.blogspot.com/2008/03/how-fast-is-java-volatile-or-atomic-or.html — сравнение скорости различных способов синхронизации

Что умеет JVM?

- Biased locking
- Lock coarsening
- Lock elision (escape analysis)

http://www.infoq.com/articles/java-threading-optimizations-p1 — Do Java threading optimizations actually work?

Что умеет JVM?

Biased locking (new in 1.5.0_6):

- Большинство объектов за всю жизнь блокирует только 1 нить
- Делается так, чтобы для этой нити синхронизация была *очень* быстрой (даже без CAS)
- `-XX:+UseBiasedLocking` (по умолчанию включено начиная с Java SE 6)

`http://tinyurl.com/biasedlocking` — от разработчиков JVM, по-русски

Что умеет JVM?

Lock coarsening:

```
public void setupFoo(Foo foo) {
    foo.setBarCount(1);
    foo.setQuxEnabled(true);
    foo.addGazonk(new Gazonk());
}
```

...и все 3 метода `synchronized`.

Тут незачем делать `synchronized` 3 раза. Можно запихнуть все в 1 `synchronized`.

Опция `-XX:+EliminateLocks` (по умолчанию включено).

`http://citeseer.ist.psu.edu/459328.html` — статья про технику.

Что умеет JVM?

Escape analysis: Если объект не убегает в другую нить, то нет смысла делать на нем `synchronized`.

```
DeepThought dt = new DeepThought();
dt.turnOnPower();
dt.enterData(data);
dt.wakeup();
return dt.computeUltimateAnswer();
```

Опция `-XX:+DoEscapeAnalysis`.

Тестирование

Тестировать многопоточный код адски сложно.

- Протестировать правильность детерминированного кода — можно, хоть и трудно
- Протестировать отсутствие багов, связанных с недетерминизмом — совсем сложно
- Нет контроля над главным фактором трудноуловимых ошибок — недетерминизмом шедулера
- Необходимо стресс-тестирование
  - И даже оно *ничего* не гарантирует
  - Лучше, конечно, доказывать корректность
- Ошибки все равно пролезут. Их должно быть хорошо видно.

Тестирование

Что нужно оттестировать?

- Код последовательно корректен
- Код правильно работает без нагрузки
- Код не ломается под нагрузкой

Именно в таком порядке.

Наличие ожидаемого поведения

Как проверить, что многопоточный код реализует определенный протокол в “идеальных” условиях?

- Понаставить задержек, тем самым сделав порядок событий детерминированным
- Проверить, что происходят нужные события в нужном порядке
  - Проверять, что блокирующие вызовы — блокируются
    - Измерить длительность или добавить out-параметр `didBlock`
  - Проверять, что неблокирующие вызовы — не блокируются
  - Mock objects, listeners во всех важных точках кода

Криво, длинно, уродливо, но работает.

Устойчивость к нагрузке

Как проверить, что многопоточный код не ломается под нагрузкой?

- Придумать инварианты
- Организовать нагрузку
- Постоянно проверять инварианты

Длинно, уродливо, без гарантий, но работает.

Устойчивость к нагрузке

Пример: Тестируем `ConcurrentMap`.

- Инвариант: Каждый добавленный, но не удаленный элемент присутствует в мапе
- Инвариант: `size()` равно числу добавленных, но не удаленных элементов.

Устойчивость к нагрузке

Пример: Тестируем `ConcurrentMap`

- Повторить 100000000000000 раз
  - Сгенерировать 10000 случайных значений, разбить на 10 частей случайным образом
  - В 10 нитей:
    - добавлять в мапу значения из i-й части со случайными небольшими задержками
    - проверять, что все предыдущие значения из i-й части в мапе есть
  - По окончании проверить наличие каждого значения
  - Пригодится `CyclicBarrier`

Повторяемость

Как повторить готовую ошибку?

- Написать делегирующие mock-объекты, записывающие все действия
- Дождаться ошибки
- Написать mock-объекты, “проигрывающие” заданный сценарий

Поможет найти “гонки” в протоколе. Не поможет найти низкоуровневые “гонки” (между statement'ами).

- Без мозгов не обойтись!

Как спровоцировать ошибку

Как организовать максимум недетерминизма шедулера?

- Понаставить `Thread.yield()` или `Thread.sleep(1)` в разных местах кода
- Использовать $\approx 2 * NCPU$ нитей
  - Чтобы и все CPU были заняты, а нити и вытеснялись, и не простаивали слишком долго

Что делать с произошедшими ошибками

Главное — не дать им ускользнуть.

- Не давать „огнеопасные“ вычисления внешним API — они часто глотают исключения. Обертывать логгингом.
  - ExecutorService
  - Обработчики событий в GWT
  - Полагаю, они не одиноки.
- По возможности не терять информацию о потоке исполнения

Что делать с произошедшими ошибками

Главное — не дать им ускользнуть.

- Не давать „огнеопасные“ вычисления внешним API — они часто глотают исключения. Обертывать логгингом.
- По возможности не терять информацию о потоке исполнения
  - При асинхронном вызове запомнить стектрейс инициатора: он пригодится при исключении в реакторе.
  - Артиллерия для отладки дедлоков: блокируемый ресурс хранит стектрейс захвата.
    - Проверено в боевых условиях. Безумные издержки, но редкий баг устоит.

Тулы

- TestNG имеет небольшие специальные средства
  - `@Test(threadPoolSize=5,invocationCount=10,threadMap=2,2,2,3,1)`
- MultithreadedTC (http://code.google.com/p/multithreadedtc/)
  - `assertTick(n)`, `waitForTick(n)`
  - Понаставить тиков можно только в тестовом коде
- Упомянутый J-LO, tracecheck ...