Разработка → Модели акторов 40 лет

Программирование*, Параллельное программирование*, Блог компании LLC Tik-Tok Coach

Высоконагруженные системы, построенные по модели акторов – это тренд сегодняшнего времени. Вот далеко неполный перечень статей на хабре, в которых, в той или иной степени, упоминается данная модель или одна из ее реализаций, например,1, 2, 3, 3, 4, 5, 6, 7. Есть хорошая статья в википедии, рассказывающая про акторы. К сожалению, после ее прочтения, у меня осталось много вопросов, ответы на которые я смог найти только в первоисточниках. Результаты этого обзора я и хочу представить Вашему вниманию.

Чем является модель акторов?

Акторы – это набор практик, методология разработки, архитектурный паттерн, маркетинговый ход?

В моем университетском курсе, как и у многих, понятие вычислимости определялось через машины Тьюринга и Поста. У машины есть состояние – совокупность значений всех ячеек ленты. Процесс вычислений представляет собой последовательность шагов машины, каждый из которых меняет ее состояние. Каждый шаг машины – выполнение одного атомарного неделимого действия (операции). Далее буду называть это традиционной моделью вычислений.

Такая трактовка вычислений приводит к понятию глобального времени, когда в один и тот же момент времени может выполняться одна и только одна атомарная операция. Это свойство существенным образом используется для доказательства свойств различных примитивов синхронизации в случае многопоточного программирования на однопроцессорных машинах, например, в книге Грегори Р. Эндрюс Основы многопоточного, параллельного и распределенного программирования.

Для многопроцессорных машин или распределенных систем требование глобального времени, в общем случае, неприемлемо. Следовательно, необходимо обобщить понятие вычислимости на случай параллельных вычислений. Одним из таких обобщений и является модель акторов.

С чего все начиналось?

Появление данной модели в далекие 70-е годы было обусловлено стойкой верой в то, что машины следующего поколения будут многопроцессорными, а программы обладать искусственным интеллектом. В 1973 году Carl Hewitt, Peter Bishop и Richard Steiger опубликовали статью A Universal Modular Actor Formalism For Artificial Intelligent. В этой статье они ввели понятие актора и объяснили, что многие классы приложений являются частным случаем модели акторов. Кстати, в этом году был 40-летний юбилей!

Актор – это универсальная абстракция вычислительной сущности, которая в ответ на получаемое сообщение

1. Может отправить конечное число сообщений другим акторам,
2. Создать конечное число акторов,
3. Выбрать поведение для приема следующего сообщения.

Чем принципиально акторы отличаются от традиционной модели вычислений?

Разница между ними такая же, как между телефоном и отправлением сообщения по почте. В первом случае (это вычисления на основе глобального времени), если несколько человек пытаются дозвониться на один телефонный номер, то они начинают конкурировать за доступ к общему разделяемому ресурсу – адресату. Офисные АТС, многоканальные телефоны, программное обеспечение call-центров – все это (а-ля примитивы синхронизации) нужно для того, чтобы обеспечить эффективную обработку входящих звонков. В случае же с почтовым отправлением отправитель письма (актор) просто посылает письмо адресату без каких-либо задержек из-за необходимости согласовывать свои действия с другими отправителями. Но! При этом нам неизвестно, когда получатель прочтет наше письмо.

Кто занимается развитием теории акторов?

Два крупнейших специалиста – это Carl Hewitt и его ученик Gul A. Agha. Hewitt, в основном, занимается строгим обоснованием того, что другие подходы к вычислимости являются частным случаем акторной модели, а Agha – различными приложениями для распределенных систем.

Ключевые результаты

Каждый из результатов сам по себе — это тема для отдельной большой статьи, поэтому приведу лишь резюме и ссылку на первоисточник.

События в модели акторов образуют частично упорядоченное множество. Аксиоматика этого множества под названием Ordering Laws была описана Carl Hewitt и Henry Baker в статье Actors and Continous Functionals (1977 г). Аксиом довольно много, а смысл их в том, чтобы обосновать, что модель акторов годится для использования на практике, например, что в любой момент времени количество сообщений, адресованных одному получателю, конечно.

В 1985 году Gul A. Agha под руководством Hewitt защитил диссертацию Actors: A Model Of Concurrent Computations in Distributed Systems. В диссертации Agha описывается синтаксис минимального языка программирования, поддерживающего акторы, а также набор типовых задач и приемов их решения (глава 4).

Еще одним важным вкладом в практические подходы к реализации модели акторов можно считать статью Phillip Haller и Martin Odersky Event-Based Programming without Inversion Control. В ней был предложен новый подход для разработки акторных систем, который был использован в Scala. Суть его в том, что получаемая параллельная программа по записи очень похожа на «обычное последовательное» программирование.

Такой же путь, например, был выбран для развития C#. Вот так выглядят акторы на C# 5:

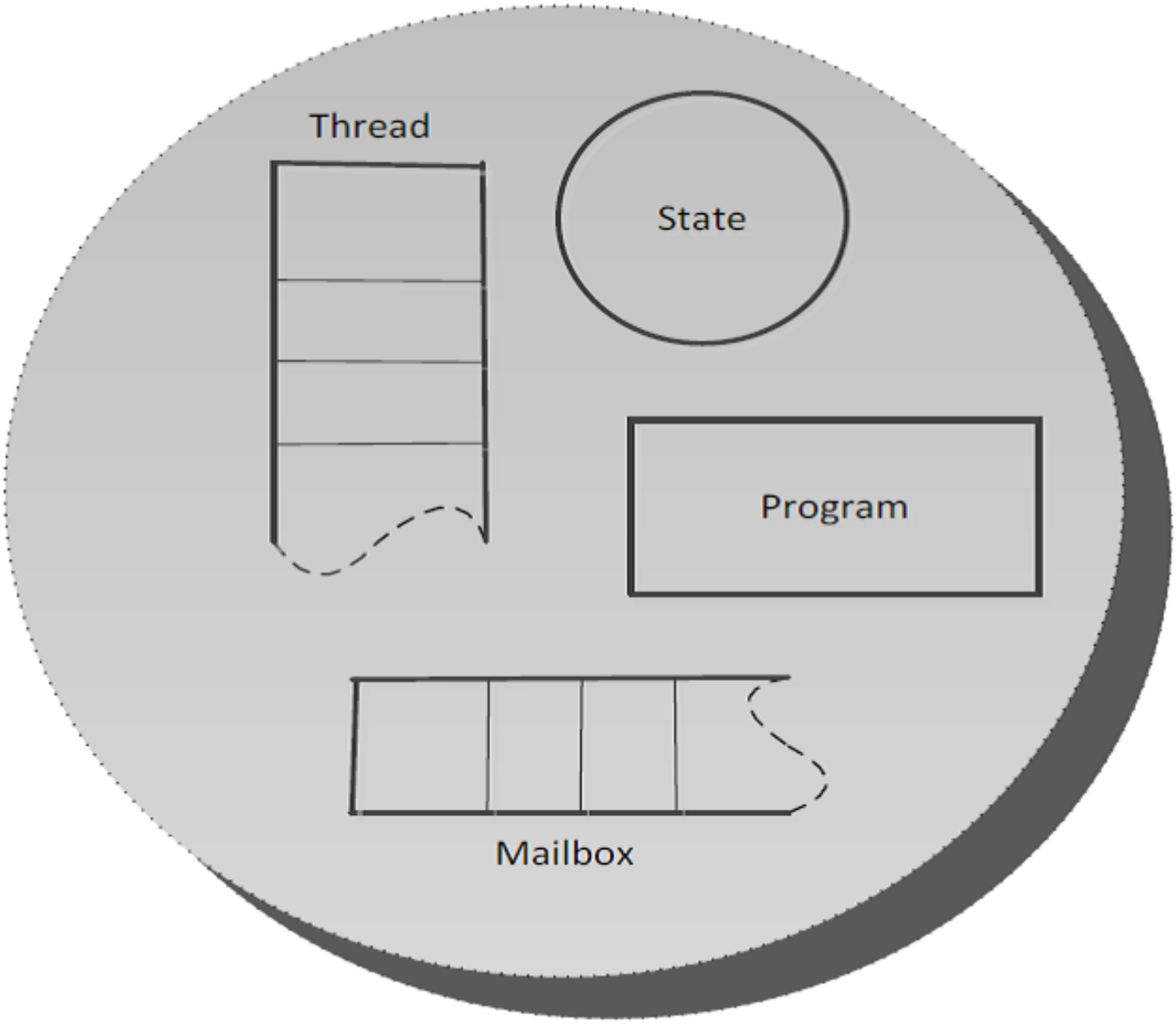
```
static async Task SavePage(string file, string a)
{
  using (var stream = File.AppendText(file))
```

```
    {
        var html = await new WebClient().DownloadStringTaskAsync(a);
        await stream.WriteAsync(html);
    }
}
```

Здесь ключевое слово **await** подразумевает асинхронный вызов соответствующего метода и возврат к вычислениям, когда будет получен ответ.

В 2011 году Gul A. Aga и Karmani подытожили многолетний опыт реализации акторных систем, описав наиболее распространенный способ реализации. Они назвали его Fog Cutter. Правда, Hewitt неоднократно критиковал такую архитектуру, например, здесь и здесь Суть критики сводится к тому, что это всего лишь одна из возможных реализаций частного случая модели, которая не реализует всю акторную модель в полном объеме.

Действительно, в phd диссертации Foundations of Actor Semantics William Duglas Clinger (ученик Карла Хьюита) показывает, что акторная модель обладает неограниченным недетерминизмом, в то время как машина Тьюринга является ограниченно недетерминированной. Из этого он и Карл Хьюит делают вывод, что существуют алгоритмы, которые можно реализовать в акторной модели, но нельзя реализовать на машине Тьюринга. Поэтому любая попытка реализации акторной модели на «обычных» компьютерах, реализующих вычислимость по Тьюрингу, приведет к тому, что будет реализован лишь частный случай акторной модели.



Ложка дегдя …

В 1988 году Роберт Ковальски, создатель языка Пролог, выдвинул тезис, что «вычисления могу быть сгруппированы по логическим выводам». Это справедливо для последовательных вычислений и для некоторых моделей параллельных. Но в 1988 году Hewitt и Agha опубликовали статью Guarded Horn clause languages: are they deductive and Logical?, в которой показали, что для модели акторов это неверно в следующем смысле: текущее состояние программы может дедуктивно не следовать из предыдущего. Что это значит на практике: отладка программы на основе модели акторов не так эффективна, как в случае последовательных программ.

Почему функциональные языки?

Почему функциональные языки возникают везде, где речь заходит о параллельном программировании? Чтобы ответить на этот вопрос, рассмотрим небольшой пример на C++:

```
int j = 0;
for(int i =0; ; ++i)
{
```

```
    ++j;
}
```

Поскольку данный кусок кода не содержит операций ввод-вывода, то, чтобы дать возможность другому потоку выполнить свой код на том же ядре процессора, требуется дождаться, когда планировщик потоков операционной системы принудительно произведет переключение потоков. Это может произойти либо, когда в другом потоке произойдет вызов системной функции, либо по событию от таймера. По умолчанию событие по таймеру срабатывает 1 раз в 15,6 мс, здесь объясняется почему не стоит уменьшать это время до 1 мс. Получается, что императивный стиль позволяет написать «жадную» программу, которая пытается единолично использовать все ресурсы процессора, а мы ограничены в средствах на это повлиять.

Перепишем эту программку через хвостовую рекурсию.

```
void cycle(int i, int& j)
{
        ++j;
        cycle(i+1, j);
}
```

Да простят меня почитатели функциональных языков программирования за такую вольную интерпретацию, но моя цель – только продемонстрировать идею.

Итерация цикла заменена на вызов функции. Предположим, что в точку вызова каждой функции компилятор встраивает код для подсчета времени, затраченного на выполнение текущего потока и переключение на другой поток в случае необходимости. Теперь у нас появляется возможность переключиться с одного потока на другой в течение наносекунд. Эта же идея позволяет реализовать легковесные потоки, например, как в Erlang.

За эти свойства функциональные языки удобно использовать там, где нужна параллельность и реакция в режиме реального времени.

Реабилитация императивных языков

Если нужен очень быстрый отклик, то функциональные языки вне конкуренции, но что если во время обработки запроса нам приходится обращаться в базу данных, а время отклика от нее составляет порядка 50-100 мс, или приходится выполнять много вычислений, то есть совершать много вызовов функций. Накладные расходы, связанные с подсчетом времени выполнения и переключением потоков на каждый вызов, дают о себе знать, и императивные языки оказываются в этом случае эффективнее. Чтобы убедиться в этом, достаточно посмотреть сайт benchmarksgame.alioth.debian.org. Сайт предлагает измерять производительность программ, написанных на разных языках, сравнивая решения одной и той же задачи. Вот, некоторые примеры сравнений: reverse-complement, mandelbrot, regex-dna, pidigits.

Сразу оговорюсь, во-первых, это только тесты – не надо к ним относиться очень серьезно. С другой стороны, любой желающий, недовольный положением своего любимого языка программирования, может предложить собственное решение и поменять расстановку сил. Во-вторых, я привел только те ссылки, где императивные языки выигрывают за явным преимуществом, потому что моя цель – лишь проиллюстрировать идею, высказанную несколько абзацев раньше, о накладных расходах, связанных с организацией параллельности.

Еще одно интересное наблюдение – реализации модели акторов на императивных языках, как правило, в тестах выше функционального Erlang. Опять оговорюсь, все знают, что Erlang хорош не в вычислениях, а именно, там, где нужен мгновенный отклик, но это только лишний раз подтверждает мысль о накладных расходах.

Приведу одну метафору: есть бегуны на длинные дистанции – марафонцы, а есть – на короткие – спринтеры. От одних требуется выносливость при невысокой средней скорости (относительно спринтеров), а от вторых – максимальная производительность в очень короткое время (относительно марафонцев). Увы, но, просто физиологически, невозможно показывать одинаково высокие результаты на спринтерских и марафонских дистанциях, просто потому, что от мышц требуются совершенно разные свойства. Эти бегуны отличаются даже фигурами: марафонцы – сухие и жилистые, а спринтеры – атлетичные и мускулистые.

5-ое поколение ЭВМ

Еще одним подтверждением, что с параллельным программированием не все так однозначно, является проект по созданию 5-го поколения ЭВМ.

В 80-х годах прошлого века в Японии была предпринята попытка создания компьютеров 5-го поколения. Проект был инициирован опять верой в то, что будущее за параллельными вычислениями, большими базами данных и логической обработкой больших баз данных.

Было затрачено 500 млн. долларов в ценах 80-х. Проект длился 10 лет и завершился полным провалом! Параллельные программы не дали существенных преимуществ в производительности над однопоточными. И на десятилетия пальму первенства захватила компания Intel со своими однопоточными процессорами. И только, когда производители процессоров уперлись в технологические ограничения по наращиванию тактовых частот, идея параллельности вновь начала набирать популярность.

Подходы к реализации модели акторов

Вопросы производительности затронутые выше, а также то, что Hewitt настойчиво делает акцент на том, что ни потоки, ни очереди сообщений, ни какие-либо известные конструкции не являются частью модели акторов привело к многообразию идей и способов реализации.

Выделяются три направления:

- Новый язык программирования. Эта ниша занята функциональными языками. Яркий пример: Erlang.
- Расширения существующих языков. Например, Scala, C#. Используется идея Odersky и Haller, которая позволяет писать параллельные программы в «привычном стиле».

- Библиотеки для существующего языка.

В заключение два наблюдения

Вера в параллельные вычисления.

Читаешь статьи 70-х годов: будущее за многопроцессорными системами, искусственным интеллектом; 80-годов: будущее за многопроцессорными системами, большими базами данных, логической обработкой данных, сейчас: будущее за процессорами с большим количеством ядер, большими данными, облачными вычислениями, интеллектуальным анализом данных. Нетрудно провести аналогии между нынешней ситуацией и прогнозами 40-летней давности. Все это напоминает пример из тренингов по мотивации про ослика и морковку. Мы делаем шаг, чтобы приблизиться к заветной цели, а цель от нас отдаляется настолько, насколько мы к ней приблизились, но именно она заставляет нас развиваться и двигаться вперед.

Игнорирование с одной стороны и критика с другой.

Когда первый раз натолкнулся на модель акторов, а это было относительно недавно. У меня возник вопрос – почему про нее не узнал раньше. Специально посмотрел книжки Таненбаума, Грегори Эндрюса, Паттерны интеграции корпоративных приложений и др. – все они довольно много места отводят концепции обмена сообщениями, но никто из них ни слова не говорит про модель акторов и Hewitt. Зато, когда читаешь Hewitt или Agha – достаточно много времени отводится критике традиционной модели вычислений. Такое ощущение, что акторы просто игнорируют. Так же происходит и в программировании: в мире императивного программирования практически ничего неизвестно о функциональном – его как бы нет, по крайней мере, это было до недавнего времени, а в мире функционального программирования императивное подвергается критике. Может есть у кого мысли, почему так произошло?

Что дальше?

Если данная статья вызовет интерес, то планирую написать серию статей, например:

- Законы упорядочивания акторов по Actors and Continous Functionals.
- Обзор идей, изложенных в диссертации Actors: A Model Of Concurrent Computations in Distributed Systems Gul A. Agha.
- Плюсы и минусы подхода Odersky Event-Based Programming with Inversion Control.
- Возвращаясь, к ложке дегдя… — пару собственных идей, уже внедренных и проверенных на практике, как можно повысить надежность и отказоустойчивость систем, реализованных по модели акторов.

Если будут пожелания по статьям, то рад буду услышать.