Специально для Хабра
https://habr.com/ru/post/589667/

Каминский Илья
Декабрь 2021

Алгоритм работы планировщика задач

Резюме

Настоящий документ описывает принципы работы этого планировщика задач и помогает читателю лучше понять положенные в основу алгоритмы.

Этот документ позволит вам самостоятельно строить высокопроизводительные планировщики задач с возможностью шага в миллисекунды или микросекунды, когда вам понадобится это сделать для систем реального времени или высоконагруженных систем.

Настоящий планировщик использует григорианскую календарную систему, как общемировой стандарт. Если вы проектируете планировщик для иной календарной системы, см. раздел «високосные дни», чтобы получить соответствующие рекомендации.

Лицензия и авторские права

Введение

Задача планирования и запуска неких процессов возникает перед разработчиками не впервые. В среде Unix наиболее популярен планировщик cron, который позволяет пользователю задавать в простом текстовом виде расписания запуска различных процессов, сценариев и утилит. В системах семейства Windows также есть аналогичные утилиты. Однако именно формат описания расписаний от cron сыскал у разработчиков наибольшую популярность и вышел далеко за пределы экосистемы Unix.

Обычно, задача перед планировщиком ставится простая: по известному расписанию, произвольно задаваемому пользователем в текстовом виде, необходимо определять ближайшие в будущем или прошлом даты следующих событий. Т.е. от текущей даты (или от даты, задаваемой на входе), планировщик должен найти ближайшую по расписанию.

Оригинальный cron, применяемый в мире Unix, может планировать задачи только с шагом в минуту. Такой частоты вполне достаточно для планирования запуска приложений пользователя. Но в некоторых программных системах возникает необходимость в планировании задач с большей разрешающей способностью: до секунд, миллисекунд, и даже микросекунд.

Такие требования с одной стороны усложняют код, делая традиционный подход крона нерентабельным, с другой – накладывают более жесткие рамки и на его время выполнения. Ведь зачастую, результат работы планировщика – это один из этапов обработки некого бизнес-

процесса. И вызывающий планировщик код должен получать достаточное количество времени на свою работу. Иными словами, такой планировщик задач должен работать за время гораздо более меньшее, чем минимально возможный его шаг. И если вы гарантируете, что можете планировать задачи с шагом в микросекунду, то ваш планировщик должен отрабатывать за считанные наносекунды, а то и вовсе считанные такты процессора.

В некоторых случаях, разработчиками накладываются дополнительные ограничения на объём используемой планировщиком памяти. В тех ситуациях, когда в программной системе планируется запуск большого числа планировщиков для разных бизнес-процессов.

Настоящий документ описывает, как добиться максимальной производительности при минимальном потреблении памяти.

Формат планировщика

Полный формат расписаний выглядит как «yyyy.MM.dd w HH:mm:ss.fff», где каждый элемент, это соответственно год, месяц, день месяца, день недели, часы, минуты, секунды, миллисекунды. Всего 8 компонентов. Более подробно формат описан в описании к проекту.

Но нам важно другое: каждую часть даты/времени можно задавать в виде списков диапазонов и интервалов с шагом. Включая наличие специального символа * (звездочка), означающего «любое значение», мы имеем 5 видов элементов списка:

- Константа, например: 2
- Диапазон, например: 1-30
- Интервал с шагом, например: 10-40/3
- Звездочка, например: *
- Звездочка с шагом, например */3.

Само же расписание компонента даты/времени – это совокупность (список) таких элементов. Например, расписание только для минут может выглядеть так: «0-10/3,15,20,22,40-59». И так для каждого компонента даты/времени независимо друг от друга. Т.е. в теории расписание может быть достаточно сложным. И необходимо найти оптимальный подход к хранению расписания в удобной для планировщика форме.

Техника оптимального хранения расписания

Очевидно, что текстовый вариант расписания не очень удобен для работы планировщика. Тем более, когда речь идет о построении высокопроизводительного кода.

Чтобы решить задачу поиска ближайшего события, необходимо расписание проанализировать и предоставить планировщику в удобном «цифровом» виде. Наиболее логичный подход – представление расписания для каждого компонента даты/времени в виде массива элементарных диапазонов с шагом.

В коде за это отвечает класс Range. А список диапазонов – RangeList. Таким образом, модель расписания планировщика представляет собой набор из 8-ми таких RangeList.

Задачу парсинга расписания и построения модели решает класс Parser. На выходе он возвращает класс ScheduleModel, хранящий 8 объектов RangeList для каждого компонента даты/времени.

Однако сама модель расписания, как ни странно, не хранится далее. На основе нее строятся матчеры – специальные объекты (каждый для своего компонента), которые будут проверять соответствие заданного времени расписанию и помогать искать следующее значение.

Матчинг компонента даты

Поскольку исходное расписание представлено покомпонентно, работать чисто с timestamp'ами не получится. Входящую дату, от которой и ведутся расчеты, придется распаковать из формата unixtime в массив. А уже потом, получив результат, упаковать обратно.

У неопытного разработчика возникает соблазн не заморачиваться с такими сложностями, а просто прибавить к текущей дате заданное число миллисекунд, чтобы получить дату следующего расписания. Но для начала здесь необходимо найти это «заданное число миллисекунд». И ещё убедится, что «текущая» дата соответствует расписанию.

Но нельзя говорить, что такая техника невозможна вовсе. В случае крайне простых расписаний и в режиме «генерации серии следующих событий от заданной даты» мы можем прибавлять фиксированные значения. Например, для расписания «*:*:*.100,150,170» возможно определить кольцевой буфер из 3-х смещений: +50, +120, +930, и последовательно прибавлять их к исходной дате в формате unixtime. В этом случае, распаковка даты не потребуется.

Очевидно, что, чем сложнее расписание, тем больше становится ваш буфер. В какой-то момент выгода от его использования станет меньше, чем выгода от алгоритма, который предлагается ниже.

И кроме того, такая техника возможна, только если исходная дата, которую вы модифицируете, в точности соответствует расписанию. Помимо этого, такое решение является ещё и «statefull». И может не подойти в ситуации, когда ваш планировщик вызывается одновременно из нескольких потоков.

Теперь, когда мы закрыли вопрос: «а почему бы просто не работать с исходным unixtime-представлением времени», определим стратегию работы с компонентами даты/времени.

Нам необходимо:

- проверять, что текущий компонент соответствует расписанию;
- возвращать следующее большее или меньшее значение, либо сообщать, что такого значения нет;
- возвращать начальное значение для компонента (минимально возможное по расписанию, или максимально возможное по расписанию).

Второй и третий пункт можно объединить, если возвращать флаг переполнения. Но в настоящей библиотеке был определен следующий интерфейс:

```
interface DigitMatcher
{
    boolean match(int value);
    boolean isAbove(int value);
    boolean isBelow(int value);
```

```
    int getNext(int value);
    int getPrev(int value);
    boolean hasNext(int value);
    boolean hasPrev(int value);
    int getLow();
    int getHigh();
}
```

Очевидно, что сложность расписания диктует разный подход к работе с компонентами.

Для расписания, заданного звездочкой, достаточно просто проверять соответствие компонента его минимальному и максимальному значениям, а в качестве следующего / предыдущего значения возвращать на единицу больше / меньше и флаг переполнения.

Не сильно усложняется логика, когда у нас одна единственная звездочка с шагом. Например, */5. В этом случае проверка на соответствие сводится к операции получения остатка при делении на 5. Он должен быть равен 0. А чтобы вернуть следующее значение достаточно прибавить или вычесть 5 (шаг) к текущему значению. Но вначале, необходимо привести текущее значение к числу, нацело делящемуся на шаг.

Одиночный интервал с шагом a-b/n и простой диапазон a-b ничем принципиальным здесь не отличаются, за исключением допустимых границ. В отличие от звездочки, эти границы задаются пользователем, тогда как для звёздочки, они определяются возможными значениями самого компонента даты/времени.

Наконец, если наше расписание – это простое число, то тут всё ещё проще. Проверка на соответствие сводится к проверке на равенство заданному числу, а при попытке вернуть большее или меньше значение, сигнализируем о переполнении.

Обратим читателя внимание, что такие простые ситуации имеют алгоритмическую сложность O(1) при проверке и O(1) при поиске. А значит, работают максимально быстро. Это идеал.

Матчинг списков

Куда сложнее с расписаниями, которые являются списками интервалов и диапазонов.

Массивы

У большинства программистов возникает соблазн использовать во всех ситуациях простой массив: в каждой ячейке отмечаем флагом допустимые расписанием значения компонента, и используем его для матчинга.

Способ, основанный на массивах, использует оригинальный код cron. Но у него не сложное расписание, и относительно большой запас времени на выполнение. Так что он не тратит много памяти, и работает приемлемо быстро. Но если вы планируете построение высокопроизводительного кода, вам стоит рассмотреть и другие методики.

Во-первых, массив расходует больше памяти. Для минут и секунд нам понадобится 60 ячеек. Для года – уже 100, а для миллисекунд – 1000. И если вы планируете добавить микросекунды и, даже, наносекунды, вам понадобится ещё пара таких же массивов по 1000 элементов каждый. Прибавьте сюда необходимость в вашем проекте использовать много экземпляров планировщика, и вы увидите, чем плох такой подход.

Во-вторых, алгоритмическая сложность поиска следующего значения для массива составляет O(n). Не подходит для скоростных приложений. Так что, простой массив – это не самый оптимальный способ.

Деревья

Следующий подход, который предпочтут более опытные программисты, – использование древовидных структур. В этом случае, как они полагают, они получают оптимальный расход памяти и оптимальное быстродействие. Однако это не совсем так.

Действительно, поиск следующего значения составляет O(log n). Но и проверка на соответствие текущего значения заданному расписанию тоже составляет O(log n). Т.е. мы получаем результат немного худший, чем у массива. К тому же, с ростом сложности расписания (увеличением числа допустимых значений), массив будет выигрывать и по скорости работы и по занимаемой памяти.

Последнее обеспечивается более компактным расположением элементов. Тогда как дереву на каждый узел необходимо хранить дополнительную информацию (указатели, флаги и т.п.), что сводит на нет преимущество перед массивом с ростом числа допустимых значений.

В скорости массив также начнет выигрывать из-за того, что среднее число проверок при плотном заполнении допустимых значений будет стремиться к 1. Отсюда и алгоритмическая сложность поиска (в силу мультипликативности функции O) будет стремиться к O(1).

Значит массив лучше дерева? На этот вопрос нет однозначного ответа, ибо как показано выше – это зависит от ситуации. Однако реальные замеры показали, что дерево сильно проигрывает в производительности простому массиву по всем параметрам, из-за накладных расходов по работе с памятью и из-за более сложного внутреннего устройства. Не используйте дерево, если хотите добиться максимальной производительности.

Хеш-карты

Как было сказано выше, простой массив – не самая оптимальная по памяти структура. Тем не менее, с ростом сложности расписания, и увеличения числа допустимых значений, массив более оптимален, чем дерево и другие структуры. Единственный минус – алгоритмическая сложность поиска O(n).

Чтобы сделать поиск в массиве более эффективным мы можем завести два дополнительных массива, которые сразу будут отвечать на вопрос: «кто следующий?». Тогда ответ на этот вопрос мы будем получать за константное время O(1). Именно по такому принципу исполнен класс ArrayMatcher.

Однако мы по прежнему не решили вопрос с памятью. Очевидно, что такой класс возможно использовать только для месяцев, дней, часов, минут и секунд. Для годов и миллисекунд этот класс слишком расточителен.

Если в расписании задано слишком много значений (свыше 60), то более компактным способом хранения расписания является – битовая карта. Нам понадобится 125 байт для миллисекунд и 13 байт для лет. И хотя сложность поиска в такой карте составляет O(n), она будет стремиться к O(1) по мере роста допустимых значений в расписании.

Сортированный список

Некоторые программисты захотят применить комбинацию из простых матчеров. Например для проверки компонента даты по расписанию «7,10-20», кажется логичным использовать два простых матчера. Один будет проверять на соответствие числу 7, другой – на попадание в диапазон 10-20. Так экономится память и обеспечивается константное O(1) время выполнения по всем операциям.

Однако здесь кроется коварное O(n): с ростом списка интервалов нам придется увеличивать и число матчеров, которые будут их обслуживать. Отсюда и падение в производительности. Попробуйте представить в таком стиле обработку расписания вида: «1,4,7,10,18,20-30,10-15». И это только один компонент даты.

Кроме того, такой подход потребует ещё и предварительной сортировки и объединения пересекающихся интервалов; выполнения кода матчеров в правильной последовательности. В противном случае логика работы вашего приложения будет отличаться от ожидаемой.

Тем не менее, нельзя говорить, что такой подход не имеет право на жизнь. При определенных условиях и как показали специальные замеры такая методика может сильно обогнать массив.

Такой подход имплементирует класс ListsMatcher. Но он требует определенных условий: интервалы должны быть отсортированы в списке по возрастанию и они не должны пересекаться. Т.е. требуется предварительная обработка модели расписания. В противном случае логика работы данного матчера будет неверной.

В настоящем проекте за оптимизацию расписания отвечают функции sort и optimize класса RangesList.

Виды матчеров

Учитывая сказанное выше, логичным решением будет разработать несколько различных матчеров, наиболее подходящих под тот или иной тип расписания, и применить технику ООП.

Для расписания заданного одной единственной константой в настоящем коде определен класс ConstantMatcher.

Для расписаний вида «диапазон», «интервал с шагом», а также «звёздочка», определены матчеры: IntervalMatcher (интервал без шага) и SteppingMatcher (интервал с шагом). Разделение здесь было выбрано по соображениям оптимизации.

Вы можете объединить эти три матчера в один класс и отказаться от ООП: вызов функции интерфейса обходится дороже, чем обычный прямой вызов. К такой технике могут прибегнуть программисты низкоуровневых языков: C/C++, D, Go. Но данный проект основан на Java и поэтому такая техника оптимизации здесь недоступна.

Для расписаний, состоящих из списка, решено было сделать 3 типа матчера:

- основанный на массивах ArrayMatcher – для всех компонентов даты, разброс значений которых, не превышает 64. Сложность матчинга – O(1), а поиска – O(1). Максимальный размер памяти – 144 байта.
- основанный на битовой карте BitMapMatcher – для миллисекунд и годов. Это классический массив, упакованный в битовую карту. Сложность матчинга – O(1), а поиска – O(n). Максимальный занимаемый размер – 144 байта.

- основанный на списке диапазонов ListMatcher – для миллисекунд и годов. Это комбинация простых матчеров. Применяется, когда количество диапазонов в списке не превышает 10. Максимальный занимаемый размер – 124 байта.

Теперь, когда определен состав матчеров, мы можем переходить к описанию алгоритма проверки даты и поиска следующего значения.

Алгоритм проверки даты

Наилучший способ быстро определить, подходит ли заданная дата под расписание, это применить подход, используемый в позиционных системах исчисления.

Если записать компоненты даты/времени в формате YMDhmsn (год, месяц, день, час, минута, секунда, миллисекунда) – можно получить конструкцию, напоминающую число в позиционной системе счисления. А каждый компонент даты интерпретировать как «разряд» этого числа.

Имплементация данного подхода отражена в классе CalendarDigits.

Отличие от классических позиционных систем состоит в том, что каждый «разряд» имеет свое собственное основание (количество возможных значений): месяцы – 12, часы – 24, минуты и секунды – 60. А кроме того разряд, соответствующий «дню месяца», меняет свое основание в зависимости от «разрядов» года и месяца.

При таком взгляде на даты, можно легко применить классические правила сравнения чисел, чтобы быстро определить: подходит ли данная дата под расписание или нет.

Для этого достаточно начать сравнение с самого старшего (левого) разряда. Таким разрядом будет вначале год, затем месяц, и так до миллисекунд. При любой неудачной проверке, останавливаемся и «сбрасываем» более младшие разряды на условный «ноль».

Если мы ищем в расписании дату более старшую, чем текущая, то сбрасываем младшие разряды на минимальное значение. Если мы ищем более раннюю дату – сбрасываем их на максимальное значение.

Итого, нам понадобится максимум 7 проверок по числу разрядов числа.

Такой подход дает нам главное преимущество: если в результате проверок мы не добрались до самого младшего разряда (миллисекунды), то на выходе мы сразу же получаем требуемую дату. И планировщик может смело её возвращать.

Если же входная дата полностью соответствует расписанию, мы выполним все 7 проверок, дойдя до миллисекунд. А дальше, нам необходимо будет провести алгоритм получения следующей даты.

Алгоритм получения следующей даты

Попробуем вновь посмотреть на дату как на число в нестандартной позиционной системе счисления.

Если текущая (входная) дата соответствует расписанию, то следующая дата будет на условную «единицу» больше / меньше.

Если вспомнить, как мы производим операции прибавления единицы к обычным числам позиционных систем счисления, то можно применить подобный алгоритм и к датам.

Начинаем увеличивать (или уменьшать если мы идём в обратную сторону в поисках более ранней даты) самый младший разряд. Таким у нас являются миллисекунды.

1. Если для текущего разряда в расписании существует следующее значение (здесь нам помогут функции getNext() / getPrev() матчеров), устанавливаем его, сбрасываем более младшие разряды на условный «0» и завершаем работу. Полученная дата и есть искомый результат. Если в расписании нет следующего значения, выполняем шаг 2.
2. Переходим к более старшему разряду. Повторяем шаг 1.

В самом худшем случаем нам придется выполнить 7 «инкрементов» от миллисекунд до секунд.

Данный алгоритм реализован в функции increment() класса CalendarDigits.

Високосные дни

Описанные выше алгоритмы не учитывают важных факторов: високосные дни и дни недели.

Полученная на выходе дата запросто может не соответствовать дню недели, определенному расписанием. И нам придется начинать поиск сначала.

А при попытке «инкремента» месяца и сброса дня на условный «0» мы запросто можем получить 29 февраля в не високосном году, или 31 апреля. А кроме того, в расписании ещё предусмотрена специальная константа 32, означающая «последний день месяца». Поэтому, мы должны следить за тем, чтобы матчер нам не порекомендовал значение 32 в качестве следующего.

Как эффективно искать подходящие дни недели мы опишем в главе ниже. А пока попробуем решить проблему дней месяца.

Попробуем немного модифицировать алгоритм «сброса» и «инкремента» даты. При сбросе младших компонентов даты мы движемся вниз к миллисекундам, при инкременте – мы движемся вверх к самому старшему разряду (годы).

Путь в момент попытки установки дня месяца нам известно, является ли этот день действительным с точки зрения календарной системы (он действителен пока только с точки зрения расписания). Если день правильный – продолжаем двигаться вверх или вниз как и раньше. Если нет – то переключаемся на алгоритм «инкремента даты» со старшего разряда.

Переходим к разряду выше (это месяц) и инкрементируем его (декрементируем, если ищем предыдущую дату). Это вынудит нас вернутся к алгоритму «сброса» младших компонентов даты. Первым младшим компонентом будет день месяца. Если после «сброса» он снова не удовлетворяет календарной системе (например для расписания 30-31), вновь повторяем алгоритм с начала настоящего абзаца. Если удовлетворяет – продолжаем алгоритм «сброса». Далее проблем не будет.

Именно такой подход реализует функция tryToSetupDayOfMonth класса CalendarDigits.

Чтобы матчер дня месяца не возвращал значения, не удовлетворяющие календарной системе, а также, чтобы обрабатывать специальную константу 32 («последний день месяца»), был введен специальный проки-матчер LastDayOfMonthProxy. Он «проксирует» все запросы к

оригинальному матчеру дня месяца и контролирует корректность результата в соответствии с системой Григорианского календаря.

Если вы будете разрабатывать вариант библиотеки под иную календарную систему (Лунный календарь), вам необходимо написать свою версию прокси-матчера для дней месяца, либо отказаться от неё (Юлианский календарь). Для Еврейской календарной системы вам понадобится создать прокси-матчер и для месяцев, а также сделать функцию tryToSetupDayOfMonth более универсальной. С учетом её текущей реализации правке потребуется 2 строки с вызовом: calendar.isCorrectDay. Замените на свой вариант.

Обратите внимание, что поиск правильного дня месяца вынуждает выполнять де-факто цикл. Это скрытый (третий) цикл данного алгоритма. Вы должны учитывать это, когда проектируете системы реального времени.

Поиск дней недели

Теперь, когда планировщик событий возвращает корректные даты как с точки зрения расписания, так и с точки зрения используемой календарной системы, необходимо убедиться, что полученная дата выпадает на день недели, определенный расписанием.

Т.е. мы вынуждены выполнить ещё один цикл (четвертый) поиска такой даты в расписании, которая подходит под заданный день недели.

Чтобы ускорить процесс особенно в случаях очень «неудобных» расписаний, была применена техника сравнения битовых карт.

Представим каждый выбранный в расписании день недели как 7-битную карту. Условимся, что установленный бит означает выбранный день. Условимся, что младший бит 0 соответствует воскресенью, а старший 6-й бит – субботе.

Т.к. в расписании компоненты даты задаются независимо, мы можем для дней месяца составить аналогичную карту бит. Положим, что 1-е число выпадает на 0-й бит, а 2-е число – на 1-й бит и т.д. по кругу; 8-е число – на 0-й бит. Мы можем построить битовую карту для всех выбранных дней месяца.

Теперь мы можем быстро ответить на вопрос: есть ли хотя бы один день в текущем месяце, который выпадает на выбранные дни недели? Для этого вначале необходимо привести исходную 7-битную карту дней месяца к правильной карте текущего месяца. Делается это простыми циклическими сдвигами: вначале приводится к январю текущего года, затем к выбранному месяцу. Затем анализируется результат сложения AND двух карт: карты дней месяца и карты дней недели. Если получено ненулевое значение, значит в текущем месяце есть день, выпадающий на заданный день недели.

Техника сдвигов реализована в классе DaysMap, а используется в методе findBestMonth класса Schedule.

Таким же образом мы можем построить 7-битную карту дней месяца для целого года, чтобы сразу отсеивать не подходящие под расписание дней недели. Алгоритм тривиален. Реализован в методе findBestYear классе Schedule.

Генерация следующих событий

Поиск подходящей под расписание даты на основе произвольно заданной начальной – весьма расточительный процесс. Естественное желание разработчика – получить серию дат будущих событий «наперед», чтобы в будущем выдавать уже готовый результат, не тратя вычислительных ресурсов.

Это может не совсем подойти для систем реального времени, т.к. просчет серии дат «наперед» будет требовать гораздо большего времени, чем поиск одной даты, что будет приводить к существенным задержкам.

Тем не менее, генерация серии дат событий «наперед», может быть выгодной стратегией в вашем программном обеспечении.

В настоящей библиотеке был реализован специальный интерфейс ScheduleEventsGenerator, который может быть получен из текущего экземпляра планировщика методом getEventsGenerator. Эта функция возвращает мутабельный объект, который производит последовательную серию дат событий для заданного расписания и начальной даты, а также в заданном направлении (в будущее или прошлое).

Скорость работы реализованного генератора сильно превышает вычисление дат самим планировщиком (оценочно в 10 раз: до 20-50нс, против 100-200нс планировщика). Это происходит из-за устранения накладных вычислительных расходов, а также по причине того, что работа начинается с алгоритма «инкремента» последней сгенерированной даты. Т.е. шаги «распаковки» и проверки входной даты пропускаются.

Но разработчик должен помнить, что генератор – мутабельный объект с внутренним состоянием. Его не рекомендуется кешировать. И вы должны проявлять особую осторожность при работе с ним в многопоточной среде.

Ссылки

[1] Исходная статья с постановкой задачи

[2] Реализация cron для среды unix

[3] Различные календарные системы

[4] Реализация от автора документа

Связь с автором

Вы можете связаться с автором настоящей библиотеки и предложить свои оптимальные решения. Связь возможно через проект на github [4].