Special for
habr.com

I. Kaminsky
December 2021

The algorithm of the task scheduler

Abstract

This document describes the principles of operation of this task scheduler and helps the reader to better understand the underlying algorithms.

This document will allow you to independently build a high-performance task scheduler with the ability to step in milliseconds or microseconds when you need to do it for real-time or high-load systems.

This planner uses the Gregorian calendar system as a worldwide standard. If you are designing a scheduler for a different calendar system, see the section "leap days" to get the appropriate recommendations.

Copyright Notice

This code is published under a BSD license. You can use the proposed version of the scheduler both as a ready-made library and as a base for building your high-performance versions of task schedulers in a programming language that is convenient for you.

The copyright belongs to the Habra community. The code is the fruit of the collective work of an open group of programmers.

Introduction

The task of planning and launching certain processes does not arise before developers for the first time. In the Unix environment, the cron scheduler is the most popular, which allows the user to set schedules for launching various processes, scripts and utilities in simple text form. Windows family systems also have similar utilities. However, it is the cron schedule description format that has found the greatest popularity among developers and has gone far beyond the Unix ecosystem.

Usually, the task before the scheduler is simple: according to a known schedule, arbitrarily set by the user in text form, it is necessary to determine the next dates of the following events in the future or in the past. I.e. from the current date (or from the date set at the input), the scheduler must find the nearest one according to the schedule.

The original cron used in the Unix world can schedule tasks only in increments per minute. This frequency is quite sufficient for

planning the launch of user applications. But in some software systems, there is a need for scheduling tasks with a higher resolution: up to seconds, milliseconds, and even microseconds.

Such requirements, on the one hand, complicate the code, making the traditional cron approach unprofitable, on the other hand, impose a more stringent framework on its execution time. After all, often, the result of the scheduler's work is one of the stages of processing a certain business process. And the code calling the scheduler should get enough time for its work. In other words, such a task scheduler should work in a much shorter time than its minimum possible step. And if you guarantee that you can schedule tasks in microsecond increments, then your scheduler should work out in a matter of nanoseconds, or even a few processor cycles at all.

In some cases, developers impose additional restrictions on the amount of memory used by the scheduler. In those situations when a large number of schedulers for different business processes are planned to be launched in the software system.

This document describes how to achieve maximum performance with minimal memory consumption.

The scheduler format

The full format of schedules looks like "yyyy.MM.dd w HH:mm:ss.fff", where each element is, respectively, year, month, day of the month, day of the week, hours, minutes, seconds, milliseconds. There are 8 components in total. The format is described in more detail in the description of the project.

But something else is important to us: each part of the date/time can be set in the form of lists of ranges and intervals in increments. Including the presence of a special symbol * (asterisk) meaning "any value", we have 5 types of list items:

- Constant, for example: 2
- Range,for example: 1-30
- Interval with step, for example: 10-40/3
- Single asterisk, for example: *
- Asterisk with step, for example: */3.

The schedule of the date/time component itself is a collection (list) such elements. For example, a schedule for minutes only might look like this: "0-10/3,15,20,22,40-59 ". And so for each component of the date/time independently of each other. I.e., in theory, the schedule can be quite complex. And it is necessary to find the optimal approach to storing the schedule in a form convenient for the scheduler.

Optimal schedule storage

Obviously, the text version of the schedule is not very convenient for the scheduler. Especially when it comes to building high-performance code.

To solve the problem of finding the nearest event, it is necessary to analyze the schedule and provide it to the scheduler in a convenient "digital" form. The most logical approach is to present the schedule for each date/time component as an array of elementary ranges in increments.

In the code, the Range class is responsible for this. And the list of ranges is RangeList. Thus, the scheduler schedule model is a set of 8 such lists of ranges.

The task of parsing the schedule and building the model is solved by the Parser class. At the output, it returns a ScheduleModel class that stores 8 RangeList objects for each date/time component.

However, the schedule model itself, oddly enough, is not stored further. Based on it, matchers are built - special objects (each for its own component) that will check whether a given time corresponds to a schedule and help search for the next value.

Match the components of date

Since the original schedule is presented component-by-component, work with clear timestamp won't work. The incoming date, from which the calculations are carried out, will have to be unpacked from the Unix-time format into an array. And only then, after receiving the result, pack it back.

An inexperienced developer is tempted not to bother with such difficulties, but simply add a given number of milliseconds to the current date to get the date of the next schedule. But first you need to find this "specified number of milliseconds" here. And also make sure that the "current" date matches the schedule.

But we cannot say that such a technique is impossible at all. In the case of extremely simple schedules and in the mode of "generating a series of next events from a given date", we can add fixed values. For example, for the schedule "*:*:*.100,150,170" it is possible to define a ring buffer of 3 offsets: +50, +120, +930, and sequentially add them to the original date in Unix-time format. In this case, unpacking the date is not required.

Obviously, the more complex the schedule, the bigger your buffer becomes. At some point, the benefit of using it will become less than the benefit of the algorithm that is proposed below.

And besides, such a technique is only possible if the original date that you are modifying exactly matches the schedule. In addition, such a solution is also "statefull". And it may not be suitable in a situation where your scheduler is called simultaneously from several threads.

Now that we have closed the question: "why not just work with the original Unix-time", we will define a strategy for working with date/time components independently.

We need:

- check that the current component meets the schedule;
- return the next higher or lower value, or report that there is no such value;
- return the initial value for the component (the minimum possible according to the schedule, or the maximum possible according to the schedule).

The second and third items can be combined if the overflow flag is returned. But in the real library, the following interface was defined:

```
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();
}
```

Obviously, the complexity of the schedule dictates a different approach to working with components.

For a schedule specified by an asterisk, it is enough to simply check that the component corresponds to its minimum and maximum values, and return the overflow flag as the next / previous value by one more / less.

The logic is not much more complicated when we have a single asterisk with a step. For example, */5. In this case, the compliance check is reduced to the operation of obtaining the remainder when dividing by 5. It should be equal to 0. And to return the next value, it is enough to add or subtract 5 (step) to the current value. But first, it is necessary to bring the current value to a number that is completely divisible by a step.

A single interval with a step of a-b/n and a simple range of a-b do not differ in principle here, except for acceptable boundaries. Unlike the asterisk, these boundaries are set by the user, whereas for the asterisk, they are determined by the possible values of the date/time component itself.

Finally, if our schedule is a prime number, then everything is even simpler here. Checking for compliance is reduced to checking for equality to a given number, and when trying to return a larger or smaller value, we signal an overflow.

Let us draw the reader's attention to the fact that such simple situations have an algorithmic complexity of O(1) when checking and O(1) when searching. This means that they work as quickly as possible. This is the ideal.

Working with lists

It is much more difficult with schedules, which are lists of intervals and ranges.

Arrays

Most programmers are tempted to use a simple array in all situations: in each cell we mark the values of the component allowed by the schedule with a flag, and use it for matching.

The array-based method uses the original cron code [2]. But he does not have a complicated schedule, and a relatively large amount of time to complete. So it doesn't waste a lot of memory, and it works acceptably fast. But if you are planning to build high-performance code, you should consider other techniques.

Firstly, the array consumes more memory. For minutes and seconds we will need 60 cells. For a year – already 100, and for milliseconds – 1000. And if you plan to add microseconds and even nanoseconds, you will need a couple more of the same arrays of 1000 elements each. Add to this the need for your project to use many instances of the scheduler, and you will see what is wrong with this approach.

Secondly, the algorithmic complexity of finding the next value for an array is O(n). Not suitable for high-speed applications. So, a simple array is not the most optimal way.

Trees

The next approach that more experienced programmers will prefer is the use of tree structures. In this case, as they believe, they get optimal memory consumption and optimal performance. However, this is not quite true.

Indeed, the search for the next value is O(log n). But the check for compliance of the current value with the specified schedule is also O(log n). That is, we get a slightly worse result than the array. In addition, with the increasing complexity of the schedule (an increase in the number of acceptable values), the array will win both in terms of speed and occupied memory.

The latter is provided by a more compact arrangement of elements. Whereas the tree needs to store additional information for each node (pointers, flags, etc.), which negates the advantage over the array with an increase in the number of valid values.

In speed, the array will also start to win due to the fact that the average number of checks with dense filling of acceptable values will

tend to 1. Hence the algorithmic complexity of the search (due to the multiplicativity of the function O) will tend to O(1).

So an array is better than a tree? There is no definite answer to this question, because as shown above, it depends on the situation. However, real measurements showed [5] that the tree loses a lot in performance to a simple array in all respects, due to the overhead of working with memory and due to a more complex internal device. Don't use a tree if you want to maximize performance.

Hashmaps

As mentioned above, a simple array is not the most memory-optimal structure. However, with the increasing complexity of the schedule, and an increase in the number of acceptable values, an array is more optimal than a tree and other structures. The only drawback is the algorithmic complexity of the search O(n).

To make the search in the array more efficient, we can create two additional arrays that will immediately answer the question: "who is next?". Then we will get the answer to this question in constant time O(1). It is on this principle that the ArrayMatcher class is executed.

However, we still haven't solved the memory issue. Obviously, such a class can only be used for months, days, hours, minutes and seconds. For years and milliseconds, this class is too wasteful.

If too many values are set in the schedule (over 60), then a more compact way to store the schedule is a bitmap. We will need 125 bytes for milliseconds and 13 bytes for years. And although the complexity of searching in such a map is O(n), it will tend to O(1) as the allowable values in the schedule grow.

This method also uses the original cron code [2].

Ranges combination

Some programmers will want to apply a combination of simple wizards. For example, to check the date component according to the schedule "7.10-20", it seems logical to use two simple matchers. One will check for compliance with the number 7, the other - for falling into the range of 10-20. This saves memory and ensures constant O(1) execution time for all operations.

However, here lies the insidious O(n): with the growth of the list of intervals, we will have to increase the number of masters who will serve them. Hence the drop in productivity. Try to imagine in this style the processing of a schedule of the form: "1,4,7,10,18,20-30,10-15 ". And this is only one component of the date.

In addition, this approach will also require pre-sorting and combining intersecting intervals; executing the code of the matchers in the correct sequence. Otherwise, the logic of your application will differ from what is expected.

Nevertheless, it cannot be said that such an approach does not have the right to life. Under certain conditions, and as special

measurements have shown, such a technique can greatly overtake the array.

This approach implements the ListsMatcher class. But it requires certain conditions: the intervals must be sorted in ascending order in the list and they must not overlap. I.e., preprocessing of the schedule model is required. Otherwise, the logic of the operation of this matcher will be incorrect.

In this project, the sort() and optimize() functions of the Rangelist class are responsible for optimizing the schedule.

Types of matchers

Considering the above, the logical solution would be to develop several different masters that are most suitable for a particular type of schedule, and apply the OOP technique.

For a schedule set by a single constant, the ConstantMatcher class is defined in this code.

For schedules of the type "range", "interval with step", as well as "asterisk", matchers are defined: IntervalMatcher (interval without step) and SteppingMatcher (interval with step). The separation here was chosen for optimization reasons.

You can combine these three matchers into one class and abandon OOP: calling an interface function is more expensive than a normal direct call. Programmers of low-level languages can resort to this technique: C/C++, D, Go. But this project is based on Java and therefore such an optimization technique is not available here.

For schedules consisting of a list, it was decided to make 3 types of matcher:

- array-based ArrayMatcher - for all date components, the spread of values of which does not exceed 64. The complexity of the matching is O(1), and the search is O(1). The maximum memory size is 144 bytes.
- bitmap-based BitMapMatcher - for milliseconds and years. This is a classic array packed into a bitmap. The complexity of the matching is O(1), and the search is O(n). The maximum occupied size is 144 bytes.
- based on a list of ListMatcher ranges - for milliseconds and years. This is a combination of simple masters. It is used when the number of ranges in the list does not exceed 10. The maximum occupied size is 124 bytes.

Now that the composition of the masters has been determined, we can proceed to the description of the algorithm for checking the date and searching for the next value.

Test date algorithm

The best way to quickly determine whether a given date fits the schedule is to apply the approach used in positional calculus systems.

If you write down the date/time components in the YMDhmsn format (year, month, day, hour, minute, second, millisecond), you can get a construction resembling a number in the positional number system. And each component of the date is interpreted as a "digit" of this number.

The implementation of this in the `CalendarDigits` class.

The difference from classical positional systems is that each "digit" has its own basis (the number of possible values): months – 12, hours - 24, minutes and seconds - 60. And besides, the discharge corresponding to the "day of the month" changes its base depending on the "discharges" of the year and month.

With this view of dates, you can easily apply the classic rules of comparing numbers to quickly determine whether a given date fits the schedule or not.

To do this, it is enough to start the comparison with the most senior (left) digit. Such a discharge will first be a year, then a month, and so on up to milliseconds. At any unsuccessful check, we stop and "reset" the lower digits to a conditional "zero".

If we are looking for a date in the schedule that is older than the current one, then we reset the lower digits to the minimum value. If we are looking for an earlier date, we reset them to the maximum value.

In total, we will need a maximum of 7 checks on the number of digits of the number.

This approach gives us the main advantage: if, as a result of the checks, we did not get to the lowest digit (milliseconds), then we immediately get the required date at the output. And the scheduler can safely return it.

If the input date fully corresponds to the schedule, we will perform all 7 checks, reaching milliseconds. And then, we will need to carry out an algorithm for obtaining the next date.

Get the next date algorithm

Let's try to look at the date again as a number in a non-standard positional number system.

If the current (input) date corresponds to the schedule, then the next date will be a conditional "unit" more / less.

If we remember how we perform operations of adding one to the usual numbers of positional number systems, then we can apply a similar algorithm to dates.

We start to increase (or decrease if we go in the opposite direction in search of an earlier date) the youngest category. This is what milliseconds are for us.

1. If the following value exists for the current digit in the schedule (the GetNext() / getPrev() functions of the matchers will help us here), set it, reset the lower digits to the conditional "0" and shut down. The received date is the desired result. If there is no next value in the schedule, perform step 2.
2. Go to the higher level. Repeat step 1.

In the worst case, we will have to perform 7 "increments" from milliseconds to seconds.

This algorithm is implemented in the `increment()` function of the `CalendarDigits` class.

Leap days

The algorithms described above do not take into account important factors: leap days and days of the week.

The date received at the output may easily not correspond to the day of the week defined by the schedule. And we'll have to start the search all over again.

And if we try to "increment" the month and reset the day to a conditional "0", we can easily get 29th February in a non-leap year, or 31th April. And besides, the schedule also provides for a special constant "32", meaning "the last day of the month". Therefore, we must make sure that the matcher does not recommend the value "32" to us as the next one.

We will describe how to effectively search for suitable days of the week in the chapter below. In the meantime, let's try to solve the problem of the days of the month.

Let's try to slightly modify the algorithm of "reset" and "increment" of the date. When resetting the lower components of the date, we move down to milliseconds, when incrementing, we move up to the highest digit (years).

Path at the time of the attempt to set the day of the month, we know whether this day is valid from the point of view of the calendar system (it is valid so far only from the point of view of the schedule). If the day is right, we continue to move up or down as before. If not, then switch to the "date increment" algorithm from the highest digit.

We go to the category above (this is the month) and increment it (we decrement it if we are looking for the previous date). This will force us to return to the algorithm of "resetting" the minor components of the date. The first minor component will be the day of the month. If after the "reset" it does not satisfy the calendar system again (for example, for schedule 30-31), we repeat the

algorithm again from the beginning of this paragraph. If it satisfies- we continue the "reset" algorithm. There will be no further problems.

This is the approach implemented by the tryToSetupDayOfMonth() function of the CalendarDigits class.

In order for the day of the month matcher not to return values that do not satisfy the calendar system, and also to process the special constant 32 ("last day of the month"), a special LastDayOfMonthProxy proxy-matcher was introduced. It "proxies" all requests to the original matcher of the day of the month and controls the correctness of the result in accordance with the Gregorian calendar system.

If you are going to develop a version of the library for a different calendar system (Lunar calendar for example), you need to write your own version of the proxy matcher for the days of the month, or abandon it (Julian Calendar for example). For the Jewish calendar system, you will need to create a proxy matcher for months, as well as make the tryToSetupDayOfMonth function more universal. Taking into account its current implementation, the edit will require 2 lines with the call: calendar.isCorrectDay. Replace it with your own version.

Note that finding the right day of the month forces you to perform a cycle. This is the hidden (third) cycle of this algorithm. You have to take this into account when designing real-time systems.

Search for days of the week

Now, when event planner returns a valid date as from the point of view schedules, so from the point of view used calendar system, you must ensure that the resulting date falls on the day of the week specified by the schedule.

I.e. we have to do one more cycle (four) the search for such date in the schedule, which is matched by the specified day of the week.

To speed up the process, especially in cases of very "inconvenient" schedules, a bitmap comparison technique was applied.

Let's imagine each day of the week selected in the schedule as a 7-bit map. Let's assume that the set bit means the selected day. Let's assume that the lowest bit 0 corresponds to Sunday, and the highest 6th bit corresponds to Saturday.

Since the date components are set independently in the schedule, we can make a similar bit map for the days of the month. Suppose that the 1st falls on the 0th bit, and the 2nd falls on the 1st bit, etc. in a circle; the 8th falls on the 0th bit. We can build a bitmap for all the selected days of the month.

Now we can quickly answer the question: is there at least one day in the current month that falls on the selected days of the week? To do this, first you need to bring the original 7-bit map of the days of the month to the correct map of the current month. This is done in simple cyclical shifts: first it is brought to January of the current

year, then to the selected month. Then the result of adding AND of two bit maps is analyzed: the bits map of the days of the month and the bits map of the days of the week. If a non-zero value is received, it means that there is a day in the current month that falls on the specified day of the week.

The shift technique is implemented in the DaysMap class, and is used in the findBestMonth() method of the Schedule class.

In the same way, we can build a 7-bit map of the days of the month for the whole year in order to immediately filter out the days of the week that are unsuitable for the schedule. The algorithm is trivial. Implemented in the findBestYear() method of the Schedule class.

Next event generation

Finding a date suitable for the schedule based on an arbitrarily set initial date is a very wasteful process. The natural desire of the developer is to get a series of dates of future events "in advance" in order to produce a ready-made result in the future without wasting computing resources.

This may not be quite suitable for real-time systems, because calculating a series of dates "in advance" will require much more time than searching for a single date, which will lead to significant delays.

However, generating a series of event dates "in advance" may be a profitable strategy in your software.

In this library, a special ScheduleEventGenerator interface has been implemented, which can be obtained from the current scheduler instance by the getEventsGenerator() method. This function returns a mutable object that produces a sequential series of event dates for a given schedule and start date, as well as in a given direction (to the future or past).

The speed of the implemented generator greatly exceeds the calculation of dates by the scheduler itself (estimated 10 times: up to 20-50ns, against 100-200ns of the scheduler). This is due to the elimination of overhead computing costs, as well as due to the fact that the work begins with the "increment" algorithm of the last generated date. I.e., the steps of "unpacking" and checking the input date are skipped.

But the developer should remember that the generator is a mutable object with an internal state. It is not recommended to cache it. And you should take extra care when working with it in a multithreaded environment.

Ссылки

[1] The original article with the statement of the problem

[2] Cron native implementation for Unix

[3] The different calendar systems

[4] Implementation by author

[5] Benchmarks

Meets to author

You can contact the author of this library and offer your optimal solutions. Communication is possible through the project on github [4].