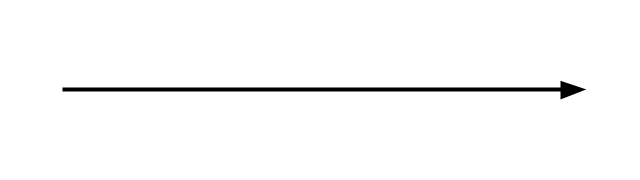
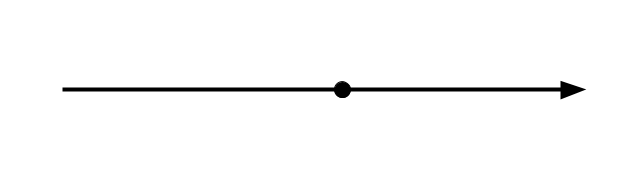
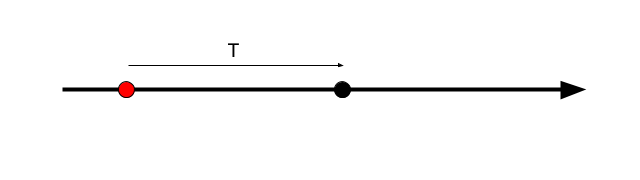
https://habrahabr.ru/post/274811/

Java и время: часть первая

* [Разработка под Linux](https://habrahabr.ru/hub/linux_dev/),
* [Программирование](https://habrahabr.ru/hub/programming/),
* [Разработка веб-сайтов](https://habrahabr.ru/hub/webdev/),
* [Java](https://habrahabr.ru/hub/java/)
* Tutorial

Восемь лет назад я принимал участие в проектировании и разработке сервиса, который был должен обслуживать запросы пользователей со всех уголков земного шара и координировать их действия. Работая над проектом я понял, что очень часто многие важные аспекты работы со временем просто игнорируются. Иногда это действительно не очень критично: если сервис локален и им пользуются только на определенной территории, либо пользователи естественным образом разделены на почти не взаимодействующие между собой географические кластеры. Однако же, если сервис объединяет пользователей по всему миру, то без четкого понимания принципов работы со временем уже не обойтись. Представим сервис, в котором общие события (совещания например) начинаются в какое-то строго определенное время, а пользователи рассчитывают на это. Какое время им показывать, в какой момент их беспокоить уведомлениями, что такое день рождения и когда можно поздравить человека — в статье я попробую это осмыслить.   
  
  
  
Статья не претендует на глубину и/или академичность. Это попытка систематизировать опыт и обратить внимание разработчиков на не очень очевидные аспекты.  
  
  
  
В JDK8 появилось новое [Date Time API](http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html), в котором появилось множество [новых полезных классов](https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html), но я не буду его упоминать, также как и об [отличных сторонних библиотеках](http://www.joda.org/joda-time/). Это отдельная, большая тема, которая заслуживает отдельной статьи; плюс, как я уже упомянул, я стараюсь рассказывать не о какой-то конкретной технологии, а о принципах работы в целом, а в этом плане новое API принципиально ничем не отличается от старого.

Временная ось

Начнем сильно издалека. Прямо очень издалека. Нарисуем ось [времени](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D1%8F).  
  
  
  
Прямо тут начинаются разные вопросы. Идет ли время всегда [в одном направлении](http://elementy.ru/lib/430702/)? Идет ли оно равномерно? Непрерывно ли оно? Что принять за единичный вектор? Едина ли эта ось для всех? Зависит ли от положения в пространстве? От скорости движения? Каков минимально измеряемый отрезок?  
  
Собственно тут я готов только задавать вопросы, но никак не отвечать на них. Есть мнение, что [никакого времени нет](http://www.newscom.md/rus/fiziki-sdelali-shokiruyushee-zayavlenie-vremeni-ne-sushestvuet.html), но я также пока не готов к обсуждению этого вопроса.  
  
Но есть и уже решенные моменты.   
  
Про [единицу измерения](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0) все ясно — она четко и однозначно специфицирована.  
  
Расстояние от Москвы до Вашингтона составляет примерно 7840000 метров и свет проходит это расстояние по поверхности земли минимум за 0.026 секунды, что совсем немало. Запрос на создание учетной записи пользователем во Владивостоке, будет отработан на московском сервере только через некоторое время. Таким образом информация о происходящих событиях доступна совсем не сразу и зависит от расстояния между точками в пространстве.   
  
Кроме того, сама скорость течения времени зависит от скорости перемещения объекта, причем даже для вполне [рядовых около-земных технологий вроде GPS](http://habrahabr.ru/company/ivideon/blog/230117/).  
  
Текущая стандартная библиотека обработки времени на Java считает, что никаких релятивистских эффектов не существует и никто не движется на около-световых скоростях, а ось времени одна и едина для всех (по крайней мере в масштабах одной планеты) — и это вполне всех нас устраивает. Возможно впоследствии в JDK #6543 будет реализован новый Java Date Time API, который позволит написать сервис для внутренней офисной системы [«Сокола Тысячелетия»](https://ru.wikipedia.org/wiki/%D0%A2%D1%8B%D1%81%D1%8F%D1%87%D0%B5%D0%BB%D0%B5%D1%82%D0%BD%D0%B8%D0%B9_%D1%81%D0%BE%D0%BA%D0%BE%D0%BB) с учетом скорости его движения и наличия/отсутствия [кротовых нор](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%BE%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BD%D0%BE%D1%80%D0%B0) рядом.  
  
Теперь отметим на временной оси некий момент. Вот, например, прямо сейчас я нажму на кнопку «точка». (Нажал)  
  
  
  
Теперь нужно придумать способ, с помощью которого я смог бы сообщить вам о том, в какой именно момент я нажал эту кнопку. Самый простой способ сделать это — обозначить какой-то момент времени, общий для нас всех, с которого мы все постоянно отсчитываем временные отсчеты. Если этот момент времени обозначен (*тот-самый-момент*), то я смогу передавать вам число своих отсчетов с этого общего момента, а вы сможете понять отношение между временем нажатия кнопки и своим текущим временем при получении моего значения.   
  
В примитивном физическом мире мы могли бы встретиться и одновременно запустить одинаковые песочные часы. После чего спокойно разойтись по своим делам, а времена событий сообщать в виде высоты песочного столба на нашем экземпляре часов (вероятно часы должны быть очень большими и громоздкими).  
  
  
  
Используемый нами *тот-самый-момент*, в свою очередь, может быть также измерением — но уже относительно какого-то более общего и важного события. В нашем случае так и происходит, *тот-самый-момент* в системе [Unix-time](https://ru.wikipedia.org/wiki/UNIX-%D0%B2%D1%80%D0%B5%D0%BC%D1%8F) (числовое значение 0), которая используется в Java — это временная точка с меткой 00:00:00 1 января 1970 от Р.Х. по UTC уже по другой шкале — [Григорианскому календарю](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B8%D0%B3%D0%BE%D1%80%D0%B8%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D1%8C).

При чем тут Java

Для задания временной точки на временной оси в Java существует класс [java.util.Date](https://docs.oracle.com/javase/7/docs/api/java/util/Date.html). Вообще, java.util.Date — это позор Java, начиная с самых ранних версий. Во-первых, у него название не отражает суть; а во-вторых, он mutable. Однако жизнь будет проще, если вы будете воспринимать его как простую обертку над внутренним полем типа long в котором хранится количество миллисекунд с *того-самого-момента* — и ничего более. Все остальные методы помечены как устаревшие и использовать их не нужно ни в коем случае. Просто запомните, что java.utl.Date тождественен (по своей сути) простому числовому long-значению Unix-time в миллисекундах. 

Date moment = new Date(1451665447567L); *// Задаем количество миллисекунд Unix-time с того-самого-момента*

moment.getTime(); *// Узнаем количество миллисекунд Unix-time с того-самого-момента.*

Если немного подумать, то становится понятно, что в любом языке есть ограничения точности представления времени. Поэтому java.util.Date (как и любые другие подобные типы) представляет собой вовсе не точку на временной оси, а отрезок. В нашем случае — отрезок миллисекундной длительности. Но с практической точки зрения такая точность нас устраивает, поэтому и дальше будем называть это точкой.  
  
Поскольку представление в Java с самого начала 64-битное, то на наш век хватит точно:

Нам хватит

Для различных операций с временем таких как чтение/установка/модификация отдельных календарных полей (год, месяц, день, часы, минуты, секунды и прочее) существует класс [java.util.Calendar](https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html). Он также не без греха — при операциях помните, что месяцы идут с 0 (лучше использовать константы), а дни идут с 1.

java.util.Calendar

Другой мутный момент в java.util.Calendar состоит в том, что при установке в нем полной даты (yyyy,MM,dd,HH,mm,ss) количество миллисекунд не сбрасывается в 0, а остается равным количеству миллисекунд от предыдущего установленного момента (текущего времени, если календарь не менялся). Поэтому, если по условиям задачи в миллисекундах должно быть 0, то это поле нужно сбросить еще одним дополнительным вызовом:

java.util.Calendar

Первое число гуляет в первых трех разрядах в зависимости от времени вызова. Это поведение может быть достаточно критичным в тестах.   
  
Для перевода временных меток в точки на оси и обратно существует класс java.text.DateFormat и его наследники.

java.text.DateFormat

Про java.text.DateFormat и java.util.Calendar обязательно нужно сказать следующее:

* У обоих классов есть метод setTimezone() для явной установки временной зоны. Крайне желательно всегда его использовать для того, чтобы обозначить, что вы полностью контролируете процесс, а не полагаетесь на временную зону по-умолчанию.
* У обоих классов есть метод setLenient() для установки «мягкого» режима. В таком режиме оба класса будут снисходительно относиться к ошибкам в календарных метках, пытаясь угадать что же вы имели в виду на самом деле. Тут зависит от ситуации, но я бы рекомендовал угадывание отключать (по умолчанию «мягкий» режим включен).
* Оба класса потоко-небезопасны. И, если для java.util.Calendar это совершенно ожидаемо (поскольку мы понимаем что у него есть внутреннее состояние), то, в случае java.text.DateFormat, это для многих оказывается сюрпризом.

Также есть (в старом АПИ) еще несколько классов:  
[java.sql.Timestamp](https://docs.oracle.com/javase/7/docs/api/java/sql/Timestamp.html) — расширение (subclass) java.util.Date с наносекундной точностью для работы с типом TIMESTAMP в БД  
[java.sql.Date](https://docs.oracle.com/javase/7/docs/api/java/sql/Date.html) — расширение (subclass) java.util.Date для работы с типом [DATE](https://en.wikipedia.org/wiki/SQL#Data_types) в БД.  
[java.sql.Time](https://docs.oracle.com/javase/7/docs/api/java/sql/Time.html) — расширение (subclass) java.util.Date для работы с типом [TIME](https://en.wikipedia.org/wiki/SQL#Data_types) в БД.  
  
  
Кроме того, любую временную точку можно хранить в виде обычного long-значения [Unix-time](https://ru.wikipedia.org/wiki/UNIX-%D0%B2%D1%80%D0%B5%D0%BC%D1%8F) в миллисекундах.

Временные зоны

Любой, кто работал в глобальной компании со множеством офисов по всему миру (или даже просто по России) знает, что информация которая содержится во фразе «совещание будет в 01 января 2016 в 14:00:00» практически бесполезна. Метка «14:00» не соответствует какой-то конкретной точке на временной оси, а вернее сказать — соответствует сразу нескольким. Для того, чтобы все собрались в видео-переговорках в одно и то же время, организатору совещания нужно указать кое-что еще, а именно временную зону, в которой мы будем интерпретировать метку «14:00». Часто временная зона подразумевается по главенству головного офиса («работаем по московскому времени»), в противном же случае, если временная зона по-умолчанию вообще никак не подразумевается, то нужно задать ее в явном виде, например «01 января 2016 в 14:00:00 MSK» — в этом случае точка на временной оси задается совершенно однозначно и все соберутся на совещание в одно и тоже время.  
  
Для устранения неоднозначности при операциях с временными метками в формате ЧЧ:MM:CC временная зона должна быть указана как при *выводе* временной метки, так и при *вводе*.  
  
Можно не указывать временную зону явно, в случаях когда ее можно каким-либо образом подразумевать неявно:

* временная зона подразумевается одной и той же по умолчанию для всего сервиса;
* временная зона явно указана в профиле самим пользователем;
* временную зону пользователя можно вычислить по его положению через гео-координаты;
* временную зону пользователя можно вычислить по его положению через IP адрес;
* временную зону пользователя можно вычислить по его положению через косвенные признаки (анализ поведения);
* текущее смещение временной зоны пользователя можно вычислить [через JavaScript](http://www.w3schools.com/jsref/jsref_gettimezoneoffset.asp);

Наверное стоит отметить, что временная зона сервиса (по умолчанию) и временная зона сервера (по умолчанию) — это в общем совсем не одно и то же. BIOS, cистема, приложение и утилиты могут, например, работать во временной зоне UTC, но при всем этом временной зоной сервиса будет временная зона Москвы. Хотя конечно сильно проще когда они совпадают — в таком случае админы настраивают временные зоны на серверах, а программисты о них не думают вообще. Если это как раз ваш случай — дальше можете не читать.  
  
Одну и ту же временную метку можно выводить по-разному для пользователей — с использованием той временной зоны, которая наиболее привычна для каждого. Например, следующие временные метки указывают на одну и ту же временную точку на временной оси:

Fri Jan 1 16:29:00 MSK 2016

Fri Jan 1 13:29:00 UTC 2016

Fri Jan 1 14:29:00 CET 2016

Fri Jan 1 21:29:00 SGT 2016

Fri Jan 1 22:29:00 JST 2016

[Википедия про временные зоны](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D0%BE%D0%B2%D0%BE%D0%B9_%D0%BF%D0%BE%D1%8F%D1%81).  
  
В Java информация о временной зоне представлена классом [java.util.TimeZone](https://docs.oracle.com/javase/7/docs/api/java/util/TimeZone.html).   
  
Нужно сказать, что временная зона — это не смещение. Нельзя сказать что GMT+3 — это Europe/Moscow. Но можно сказать, что в течение всего 2016-го года временная зона Europe/Moscow будет соответствовать смещению GMT+3. Временная зона — это вся история смещений полностью за весь исторический период, а также другие данные, которые позволяют нам правильно вычислять смещения в разные исторические моменты, а также производить правильные временные расчеты.   
  
Поисследуем временные зоны — для начала посмотрим на Europe/Moscow:

Europe/Moscow

Видим, что у зоны Europe/Moscow текущее базовое смещение составляет +3 часа относительно UTC, а общее смещение *в данный момент* составляет также +3 часа. Перевод стрелок на летнее время в перспективе отсутствует. У зоны есть отдельные имена для летнего и зимнего времени.  
  
Теперь посмотрим на парижское время:

Europe/Paris

Базовое смещение составляет +1 час относительно UTC, общее смещение *в данный момент*составляет также +1 час. Переход на летнее время в перспективе у зоны есть. Также присвоено два имени — для зимнего и для летнего времени отдельно.  
  
Теперь посмотрим на зону «GMT+5». Это фактически не совсем временная зона — у нее нет истории, нет летнего времени, а смещение постоянно.

GMT+5

Так и есть, смещение постоянно, составляет +5 часов относительно GMT и никогда не меняется. 

Примеры

Продолжим рассказывать примерами то, что слишком долго объяснять на словах. Для начала посмотрим, что происходило в 2005 году во временной зоне «Europe/Moscow»:

Europe/Moscow - 2005

Отлично, мы видим перевод стрелок на летнее и обратно на зимнее время. Посмотрим на то, что происходит в эти моменты с временными метками. Для начала — переход на зимнее время:

переход на зимнее время

Видим, что после 02:59:00 MSD стрелки сдвигаются на час назад и следующей меткой идет уже 02:00:00 MSK — зимнее время. Также временная зона говорит о том, что летнее время закончилось, а смещение изменилось с GMT+4 на GMT+3.  
  
В примере есть интересный нюанс: c помощью зоны «Europe/Moscow» совершенно невозможно установить в календаре точку соответствующую метке 02:00:00 MSD — устанавливается точка 02:00:00 MSK, что на час позже чем нужно нам. Чтобы задать эту точку как начало отсчета, приходится прибегать к услугам временной зоны UTC, в которой можно установить все. Другим вариантом может быть установка точки 01:00:00 MSD в зоне «Europe/Moscow» и прибавление часа.   
  
Теперь — переход на летнее время:

переход на летнее время

Видно, что после 01:59:00 MSK сразу следует 03:00:00 MSD — то есть перевод стрелок на час вперед. Временная зона сигнализирует, что в этот момент смещение меняется с GMT+3 на GMT+4, а также появляется флаг летнего времени.  
  
Но что будет если мы попробуем обработать метку «2005-03-27 02:30:00» в зоне «Europe/Moscow» — в теории такой метки существовать не должно?

несуществующая метка

Все верно — в строгом режиме мы получаем исключение.  
  
Посчитаем длительность дня в день перевода стрелок на зимнее время:

перевод на зимнее время

С 2005-10-30 00:00:00 MSD до 2005-10-31 00:00:00 MSK прошло 25 часов, а не 24.   
  
Теперь проверим день перехода на летнее время:

переход на летнее время

C 2005-03-27 00:00:00 MSK до 2005-03-28 00:00:00 MSD прошли 23 часа, а не 24.  
  
Эти два последних примера посвящены тем, кто прибавляет 24\*60\*60\*1000 миллисекунд не как 24 часа, а как календарный день. Вы можете сказать, теперь такой проблемы нет, так как больше нет и переводов на летнее/зимнее время. На это я могу ответить следующее:

* ваша программа должна работать корректно в любой временной зоне, а не только зоне «Europe/Moscow»;
* расчеты «назад» (в прошлое) все равно требуют корректного подхода;
* в 2016 году у нас будут выборы госдумы, а в 2018 будут выборы президента — так что я думаю, что история еще не закончена.

java.sql.Time, java.sql.Date

Типы предназначаются для работы с SQL типами [TIME и DATE](https://en.wikipedia.org/wiki/SQL#Data_types) соответственно. Подразумевается, что оба значения от временной зоны не зависят, но к сожалению это не совсем так. Поскольку оба типа являются наследниками java.util.Date — интерпретация дней-часов зависит от временной зоны:

Проблема с типами java.sql

В принципе оба типа справляются с задачей переноса информации от бизнес-логики в JDBC-драйвер поскольку обычно код и там и там работает в одной временной зоне, но в более продвинутых случаях, включая сериализацию, надо быть очень аккуратным при использовании этих классов.  
  
В новом API для соответствующих типов подобные проблемы решены.

UTC, GMT

Большинство знают, что GMT и UTC — это особые обозначения, относительно которых оформляются смещения в других временных зонах. Но не все знают, что UTC и GMT — это [не совсем одно и тоже](http://www.timeanddate.com/time/gmt-utc-time.html)(формально). Я имею в виду то, что метки «2015-12-01 00:00:00 GMT» и «2015-12-01 00:00:00 UTC» обозначают различные (хоть и близкие) точки на временной оси.  
  
[GMT](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B5%D0%B4%D0%BD%D0%B5%D0%B5_%D0%B2%D1%80%D0%B5%D0%BC%D1%8F_%D0%BF%D0%BE_%D0%93%D1%80%D0%B8%D0%BD%D0%B2%D0%B8%D1%87%D1%83) вычисляется астрономически по положению земли относительно других объектов. GMT также напрямую используется в [качестве временной зоны](http://www.timeanddate.com/time/gmt-utc-time.html) в некоторых странах.   
  
Поскольку вращение земного шара хаотично замедляется, земля оказывается в одном и том же положении через все увеличивающиеся промежутки времени. Таким образом расстояние между временными точками по соседним меткам по GMT (например «10:00:01» и «10:00:02») может точно не равняться одной [секунде](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0).   
  
[UTC](https://ru.wikipedia.org/wiki/%D0%92%D1%81%D0%B5%D0%BC%D0%B8%D1%80%D0%BD%D0%BE%D0%B5_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B2%D1%80%D0%B5%D0%BC%D1%8F) введен на замену GMT и рассчитывается по атомным часам. Непосредственно в качестве временной зоны не используется (только как опора для смещения).   
  
В UTC расстояние между временными метками (например «10:00:01» и «10:00:02») совершенно одинаковое и строго равно одной секунде. Замедление земного вращения и накапливающееся отличие от GMT решается вводом лишней секунды в году (или даже двух) — а именно [секунды координации](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%86%D0%B8%D0%B8)(leap second).  
  
Таким образом разница между точками с одинаковыми метками в GMT и UTC никогда не превышает одной секунды.  
  
[Пишут](http://astro.uni-altai.ru/~orion/blog/2011/10/gmt-ili-utc/), что время UTC практически повсюду вытеснило GMT, и что использовать обозначения смещений в виде GMT+3 уже давно моветон — правильно использовать обозначение UTC+3.   
  
Ни GMT ни UTC летнего времени не имеют.  
  
Надо сказать что [Unix-time](https://ru.wikipedia.org/wiki/UNIX-%D0%B2%D1%80%D0%B5%D0%BC%D1%8F), который используется в Java, ни UTC ни GMT напрямую не соответствует. С одной стороны в Unix-time разница между соседними метками составляет всегда равно 1 секунду, с другой стороны наличие leap second в Unix-time не предполагается.

Временная зона по умолчанию

Отображаете ли вы временную зону при выводе явно или не отображаете, запрашиваете ли вы временную зону при вводе или не запрашиваете, указываете ли вы временную зону при операциях над временем или не указываете — какая-то временная зона все равно присутствует в этих операциях неявно. Если вы не указали свою — будет использована временная зона по-умолчанию.  
  
Термин *временная зона по-умолчанию* уже был упомянут несколько раз по тексту выше. Все потому что без этого понятия ничего и объяснить толком нельзя. Все операции с временем, вывод и ввод временных меток требуют временную зону. То что вы ее не указываете, не означает что ее нет — просто она берется по-умолчанию.  
  
Но все снова не так-то просто — по умолчанию для *кого и чего*?  
  
Начнем с ядра. В [мануале к hwclock](http://linux.die.net/man/8/hwclock) сказано, что в ядре есть внутренняя концепция временной зоны, но ее почти никто не использует, кроме некоторых редких модулей — вроде драйвера файловой системы FAT. Проинформировать ядро о смене временной зоны можно этой же командой hwclock.  
  
Прикладные приложения определяют временную зону по-умолчанию несколькими способами.   
  
Во-первых, общесистемная временная зона (полная информация о ней) в Ubuntu хранится в файле (может быть симлинком) /etc/localtime, а имя этой временной зоны — в файле /etc/timezone:

$ cat /etc/timezone

Europe/Moscow

$ file /etc/localtime

/etc/localtime: timezone data, version 2, 15 gmt time flags, 15 std time flags, no leap seconds, 77 transition times, 15 abbreviation chars

$ zdump -v /etc/localtime | head -n 10

/etc/localtime -9223372036854775808 = NULL

/etc/localtime -9223372036854689408 = NULL

/etc/localtime Wed Dec 31 21:29:42 1879 UT = Wed Dec 31 23:59:59 1879 LMT isdst=0 gmtoff=9017

/etc/localtime Wed Dec 31 21:29:43 1879 UT = Thu Jan 1 00:00:00 1880 MMT isdst=0 gmtoff=9017

/etc/localtime Sun Jul 2 21:29:42 1916 UT = Sun Jul 2 23:59:59 1916 MMT isdst=0 gmtoff=9017

/etc/localtime Sun Jul 2 21:29:43 1916 UT = Mon Jul 3 00:01:02 1916 MMT isdst=0 gmtoff=9079

/etc/localtime Sun Jul 1 20:28:40 1917 UT = Sun Jul 1 22:59:59 1917 MMT isdst=0 gmtoff=9079

/etc/localtime Sun Jul 1 20:28:41 1917 UT = Mon Jul 2 00:00:00 1917 MST isdst=1 gmtoff=12679

/etc/localtime Thu Dec 27 20:28:40 1917 UT = Thu Dec 27 23:59:59 1917 MST isdst=1 gmtoff=12679

/etc/localtime Thu Dec 27 20:28:41 1917 UT = Thu Dec 27 23:00:00 1917 MMT isdst=0 gmtoff=9079

Установить временную зону для системы можно специальной командой [для вашего дистрибутива](http://www.wikihow.com/Change-the-Timezone-in-Linux), для Ubuntu это:

$ dpkg-reconfigure tzdata

А также есть вежливая утилита tzselect:

$ tzselect

Please identify a location so that time zone rules can be set correctly.

Please select a continent, ocean, "coord", or "TZ".

1) Africa

2) Americas

3) Antarctica

4) Arctic Ocean

5) Asia

6) Atlantic Ocean

7) Australia

8) Europe

9) Indian Ocean

10) Pacific Ocean

11) coord - I want to use geographical coordinates.

12) TZ - I want to specify the time zone using the Posix TZ format.

*#?*

Вторым способом указания временной зоны является переменная окружения TZ, в которой можно указать идентификатор временной зоны индивидуально для каждой программы и/или пользователя.

$ echo $TZ

$ date

Wed Dec 30 20:18:18 MSK 2015

$ TZ=UTC date

Wed Dec 30 17:18:25 UTC 2015

$ TZ=Europe/London date

Wed Dec 30 17:18:35 GMT 2015

$ TZ=Europe/Paris date

Wed Dec 30 18:18:40 CET 2015

Некоторые программы можно попросить о специфической временной зоне в настройках и/или аргументах командной строки:

$ date --utc

Fri Jan 1 08:34:36 UTC 2016

Кстати, можно попросить date вывести только текущую временную зону без времени:

$ date +%Z

MSK

Но это для обычных программ под libc, а у нас целая платформа Java. Поэтому кроме этих двух перечисленных возможностей у нас есть еще две.  
  
Можно указать аргумент для запуска JVM.

$ cat << EOF | scala -Duser.timezone=Europe/Paris

print("%s\n%s\n".format(java.util.TimeZone.getDefault().getID(), new java.util.Date()))

EOF

...

Europe/Paris

Wed Dec 30 19:24:00 CET 2015

А можно прямо в коде установить временную зону по умолчанию через метод TimeZone.setDefault(TimeZone timeZone):

$ cat << EOF | scala

> java.util.TimeZone.setDefault(java.util.TimeZone.getTimeZone("America/Los\_Angeles"))

> print("%s\n%s\n".format(java.util.TimeZone.getDefault().getID(), new java.util.Date()))

> EOF

...

America/Los\_Angeles

Wed Dec 30 10:25:45 PST 2015

Или даже все сразу:

$ TZ=Europe/London cat << EOF | scala -Duser.timezone=Europe/Paris

java.util.TimeZone.setDefault(java.util.TimeZone.getTimeZone("America/Los\_Angeles"))

print("%s\n%s\n".format(java.util.TimeZone.getDefault().getID(), new java.util.Date()))

EOF

...

America/Los\_Angeles

Wed Dec 30 10:37:28 PST 2015

База временных зон

Законодатели и [правительства](http://lenta.ru/news/2011/02/08/zimynebudet/) различных [стран](http://www.gazeta.ru/science/2015/10/26_a_7844459.shtml) и даже регионов не сидят сложа руки, регулярно включая/отключая летнее время или даже [перемещая регионы](http://lenta.ru/news/2015/11/24/time/) между [часовыми поясами](http://www.timeanddate.com/news/time/north-korea-new-time-zone.html). Критически важно иметь на системах всю последнюю информацию о подобных изменениях — в противном случае время будет вводиться и выводиться неправильно, люди будут получать СМС-ки во время своего сна, а расчеты вроде "+2 календарных дня" будут неправильными.   
  
В Linux обычные программы на libc используют базу временных зон состоящую из файлов в директории /usr/share/zoneinfo. Эти файлы принадлежат пакету tzdata, за которым активно присматривают разработчики каждого из дистрибутивов. Этот пакет своевременно обновляется и проблем с ним я не помню. В крайнем случае всю информацию можно обновить вручную, если ваша развернутая версия Linux уже больше никем не поддерживается.  
  
К счастью, мне не придется расписывать тут формат содержимого этих файлов, ни историю их возникновения — поскольку на Хабре уже есть [отличная статья на эту тему](http://habrahabr.ru/post/130401/). Не [менее отличная](https://ru.wikipedia.org/wiki/Tz_database) статья есть в википедии.  
  
Но не все так просто.  
  
Java использует свою собственную базу с временными зонами. И, если, для OpenJDK как правило можно просто и легко обновить пакет tzdata-java штатным пакетным менеджером, то для Oracle JDK придется либо [апгрейдить](http://www.oracle.com/technetwork/java/javase/downloads/index.html) всю JDK целиком на новую версию, либо пользоваться отдельной [специальной утилитой](http://www.oracle.com/technetwork/articles/javase/tzupdater-readme-136440.html) для обновления базы временных зон в уже установленной JDK.   
  
Кстати, упомянутая выше библиотеке Joda-time не использует ни системную базу tzdata, ни базу из JVM — да, у нее есть еще одна своя внутренняя база временных зон, которую нужно также обновлять [отдельным и неповторимым способом](http://joda-time.sourceforge.net/tz_update.html).   
  
Для python нужно ставить (и затем также не забыть обновлять) [отдельную библиотеку](https://pypi.python.org/pypi/pytz/).  
  
Для javascript есть куча каких-то сторонних библиотек, как минимум я точно помню что поддержка [есть в Google Closure](https://google.github.io/closure-library/api/class_goog_i18n_TimeZone.html).  
  
Вообще тема того, что тот или иной софт использует свои личные базы с временными зонами, всплывает регулярно. Например, модуль календаря Lightning для почтового клиента Thunderbird хранит свою личную sqlite-базу с часовыми зонами, и поэтому при последних изменениях в нашем государстве мне приходилось делать прямые интервенции в эту базу для корректировки. Иначе все митинги просто плыли по времени.  
  
Вообще, есть ощущение, что в основной массе разработчики не страдают паранойей (как я), про временные зоны никто не думает и в базовые поставки своих платформ tzdata никто не включает — кроме разработчиков JVM.   
  
Отдельное слово я хотел бы сказать про Android. Буду краток — временные зоны в Android это боль. При разработке платформы никто не подумал про отдельный механизм обновления tzdata, как и про то, что у законодателей по всему миру есть страшный зуд к переменам (кто бы мог подумать). Базы с временными зонами меняются только в случае, если вендор прошивки этого захочет. Учитывая то, что некоторые вендоры перестают узнавать свои собственные телефоны уже через полгода, то можно сказать, что на многих аппаратах tzdata просто не обновляется никогда. Продвинутые пользователи меняют текущую временную зону в своих аппаратах на другую, более-менее подходящую текущим условиям (например Europe/Minsk вместо Europe/Moscow). Непродвинутые пользователи все также живут в Europe/Moscow (GMT+4) и просто переводят стрелки — в результате чего временные метки событий во всех программах сдвигаются на час назад. Есть конечно вариант с рутированием и использованием сторонних решений для обновления, но всех пользователей рутировать телефоны не заставишь.

Календари

Про необходимость указания временной зоны вместе с меткой уже было сказано. Однако настоящие параноики должны бы указывать еще и систему летоисчисления. Мы не делаем этого, потому как наиболее развитая часть населения земного шара уже договорилась использовать [григорианский календарь](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B8%D0%B3%D0%BE%D1%80%D0%B8%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D1%8C), хотя мы до сих пор в удовольствием празднуем новый год по [юлианскому календарю](https://ru.wikipedia.org/wiki/%D0%AE%D0%BB%D0%B8%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D1%8C), а некоторые из нас высказывают [другие, отличающиеся точки зрения](http://www.kp.ru/daily/25655/818406/) на то, как именно правильно считать даты.  
  
Есть совершенно другие, порой достаточно странные системы счисления, в которых одно и ту же временную точку можно отобразить совершенно по-иному. Например — [календарь Чучхе](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D1%8C_%D1%87%D1%83%D1%87%D1%85%D0%B5). Вообще таких систем оказывается [достаточно много](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%82%D0%B5%D0%B3%D0%BE%D1%80%D0%B8%D1%8F:%D0%9A%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D0%B8), а все мы просто не задумываемся, что [наш календарь](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B8%D0%B3%D0%BE%D1%80%D0%B8%D0%B0%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%B0%D0%BB%D0%B5%D0%BD%D0%B4%D0%B0%D1%80%D1%8C)лишь один из многих, возможно самый используемый, но не единственный. Поиграться с некоторыми можно [тут](http://www.direct-time.ru/index.php?id=12).

Leap year

Високосный год — год в котором 366 дней, а не 365 дней как в обычном году. В високосном году добавляется один день к февралю — 29 февраля.  
  
Формула определения того, что год високосный проста и описана в [википедии](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%81%D0%BE%D0%BA%D0%BE%D1%81%D0%BD%D1%8B%D0%B9_%D0%B3%D0%BE%D0%B4)

Leap second

А вот с лишней секундой ([секунда координации](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%86%D0%B8%D0%B8)) все сильно сложнее. Суть процесса в том, что земля постоянно немножко замедляется и ее положение относительно звезд по одним и тем же меткам времени постоянно меняется. Если не производить коррекцию то время дня и ночи будет постоянно сдвигаться. Чтобы этого не происходило ученые отслеживают положение земли, вычисляют необходимую коррекцию и вносят ее в план корректировки. Поскольку процесс замедления хаотичен, долгосрочный план по коррекции составить невозможно — определение необходимости ввода секунды коррекции происходит по текущей ситуации. Также теоретически возможен ввод отрицательной секунды координации — в случае если земной шар вдруг наоборот ускорится.   
  
Подразумевается, что при наличии секунды координации, время по UTC течет с появлением 60-й секунды:

23:59:58

23:59:59

23:59:60 *# leap second*

00:00:00

00:00:01

В концепции Unix-time [не существует](https://en.wikipedia.org/wiki/Unix_time) понятия секунды с номером 60: «Because it does not handle leap seconds, it is neither a linear representation of time nor a true representation of UTC.»  
  
Для того, чтобы хоть как-то соответствовать времени по UTC [используется трюк](https://access.redhat.com/articles/15145#understanding) с переводом времени на секунду назад в полночь:

23:59:58

23:59:59

23:59:59 *# leap second*

00:00:00

00:00:01

Трюк [проводится](https://access.redhat.com/articles/15145#handling) либо через сервисы точного времени, либо самим ядром автономно на основании данных в файле таймзоны.  
  
Это именно хак, который имеет свои негативные последствия:

* Количество секунд между 23:59:00 и 00:01:00 следующего дня равно 120, а не 121 как должно быть
* Поскольку одну секунду мы съедаем, все прошлое смещается на секунду вперед

В Java, поскольку все время завязано на концепцию Unix-time, также [нет никакого учета](http://stackoverflow.com/questions/30984599/how-does-the-oracle-java-jvm-know-a-leap-second-is-occurring/30989049#30989049) leap-second. Нет ни в старом API, ни в новом API, ни в библиотеке Joda-time. При этом сама информация о leap-second в базах tzdata [есть](http://stackoverflow.com/a/30985583/827139), а в JavaDoc к методу [java.util.Date#getSeconds](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Date.java#840) говорится о том, что в неких, пока несуществующих, гипотетических Java-машинах значение поля секунд может быть равно 60 или даже 61.  
  
Сначала проверим то, что в момент leap second классы Java эту секунду не учитывают.

старое API

Результат — 120 секунд, а не 121 как должно быть.  
  
Теперь проверим новое API:

новое API

Ровно 3600 секунд, а должно быть 3601.  
  
Выясним, сколько именно секунд координации было за все время. Самое простое — проверить это на [странице в вики](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%86%D0%B8%D0%B8).  
  
Проверим и другими способами. Информация о секундах координации есть в дубликатах временных зон в директории /usr/share/zoneinfo/right.

$ file /usr/share/zoneinfo/right/UTC

/usr/share/zoneinfo/right/UTC: timezone data, version 2, 1 gmt time flag, 1 std time flag, 26 leap seconds, no transition times, 1 abbreviation char

$ zdump -v /usr/share/zoneinfo/right/UTC | grep '59:60' | wc -l

26

В основной директории /usr/share/zoneinfo файлы таймзон информацию о секундах координации не содержат.

$ file /usr/share/zoneinfo/UTC

/usr/share/zoneinfo/UTC: timezone data, version 2, 1 gmt time flag, 1 std time flag, no leap seconds, no transition times, 1 abbreviation char

$ zdump -v /usr/share/zoneinfo/UTC | grep '59:60' | wc -l

0

Как бы мы не смотрели — всего получается 26 таких секунд.  
  
Теперь посчитаем сколько секунд прошло между 1970-01-01 00:00:00 UTC и 2016-01-01 00:00:00 UTC. Посчитаем двумя способами: в Java (по Unix-time) и каким-нибудь другим, более высокоточным способом.  
  
Сначала Java:

1970-01-01 00:00:00 UTC и 2016-01-01 00:00:00 UTC - Java

Получилось 1451606400, перепроверим:

$ dateutils.ddiff --from-zone "right/UTC" -f '%S' "1970-01-01 00:00:00" "2016-01-01 00:00:00"

1451606400

Все сходится — также 1451606400 секунд, теперь натравим высокоточное оружие:

$ dateutils.ddiff --from-zone "right/UTC" -f '%rS' "1970-01-01 00:00:00" "2016-01-01 00:00:00"

1451606425

Вот оно, теперь 1451606425 секунд. Мне не очень понятно почему разница составляет 25, а не 26 секунд, но других точных калькуляторов я пока что не нашел.  
  
  
  
Становится понятно, что с путешествиями во времени есть серьезные проблемы — по крайней мере если мы делаем движок управления с использованием стандартной Java. Установив точное время, мы не сможем точно рассчитать количество секунд на которые мы должны откатиться — ошибка составит до полуминуты. Точные путешествия в будущее невозможны вообще в принципе — поскольку количество секунд координации заранее предопределить невозможно.

currentTimeMillis(), nanoTime()

Как практически на любой другой платформе в Java существует два источника времени: первый отображает текущую метку на общей временной оси, второй — считает время с момента подачи питания на процессор.

* java.lang.System#currentTimeMillis — возвращает количество миллисекунд прошедших с полночи 1 января 1970 (по временной зоне UTC). Именно этот метод используется при создании новых экземпляров java.util.Date и java.util.Calendar. Несмотря на то, что возвращаемое значение измеряется в миллисекундах, реальная гранулярность может быть сильно выше — до десятков миллисекунд. Никаких гарантий на монотонность нет, значение может скакать вперед и назад — в результате перевода системных часов оператором или сервисами точного времени.
* java.lang.System#nanoTime — возвращает некоторое абстрактное количество наносекундных «тиков». Значение тиков не обязательно берется с процессорных счетчиков — во-первых, на современном железе есть множество других источников точных сигналов; во-вторых — на многопроцессорных системах есть проблема с тем, что у каждого из процессора счетчик свой и поэтому последовательные вызовы метода могут возвращать скачущие значения в разных потоках. Конкретная реализация зависит от железа, типа и версии операционной системы. Также, несмотря на наносекундную точность результата, никаких гарантий на реальную гранулярность не дается — точность также может снижаться до десятков миллисекунд, но никак не хуже гранулярности java.lang.System#currentTimeMillis

Эти и другие вопросы подробно разобраны тут:

* [Inside the Hotspot VM: Clocks, Timers and Scheduling Events](https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks)
* [Is System.nanoTime() completely useless?](https://stackoverflow.com/questions/510462/is-system-nanotime-completely-useless/4588605#4588605)
* [Linux FAQ for clock\_gettime(CLOCK\_REALTIME)](http://juliusdavies.ca/posix_clocks/clock_realtime_linux_faq.html)

Измерение длительности операций

Исходя из из предыдущей главы, казалось бы можно сделать только один-единственный вывод о правильном измерении времени — нужно использовать только метод java.lang.System#nanoTime. Метод java.lang.System#currentTimeMillis не подходит, поскольку его значение может скакать при изменении системного времени. Этот вывод подтверждается также чтением JavaDoc к обоим методам.  
  
Тем не менее, мы смотрим в методы класса java.lang.Thread и видим нечто очень странное:

java.lang.Thread#join(long)

По всей видимости, это что-то очень древнее, поскольку, например в java.util.concurrent.ExecutorService, используется уже System.nanoTime.  
  
Этот же вопрос также очень актуален для различных сторонних библиотек, а уже сколько измерений длительности на базе System.currentTimeMillis() реализовано в самописном коде — просто не сосчитать. 

Время при тестировании

Не буду растекаться по древу в этом вопросе — просто расскажу про свой успешный опыт в уже упомянутом вначале проекте восьмилетней давности.  
  
Проект был достаточно ответственным — деньги, баллы, сложная бизнес-логика, откаты операция по таймаутам, смена статусов по истечении времени и все подобное.   
  
Разработчиков было два: я и фронтендер, тестировщиков не было вообще. На сервере у нас был фреймворк для IoC, инжекции, принципы low-coupling и high-cohesion. Не то, что бы это был необходимый запас для разработки. Но если начал читать соответствующие книги, становится трудно остановиться. Единственное что вызывало у меня опасение — это тесты. Нет ничего более беспомощного, безответственного и испорченного, чем разработчик начавший писать тесты. Но я знал, что рано или поздно мы перейдем и на эту дрянь.  
  
В общем, я сразу же решительно завел интерфейс:

org.myproject.Chronometer

Этот интерфейс инжектится практически везде, где нужно текущее время. Класс для отслеживания длительности (org.myproject.Timer) получает этот интерфейс в конструктор.  
  
Самое тяжелое при таком подходе — помнить, что можно, а что нельзя:

Можно и нельзя

Полностью отследить такие куски конечно можно только в своем личном коде, но нам важно чтобы этот правильный подход реализовывался в коде отвечающем за персистентность и бизнес-логику.   
  
При обсуждении вопросов тестирования никак невозможно не вспомнить про базы данных.

Удаление устаревших записей

При таком подходе, если он корректно реализован везде, все автотесты можно отправить в любое время — весь код можно легко выполнить в 1953 году или в 2312:

Пример теста

Если вы уже ткнули пальцем в монитор и сказали мне кг/ам за то, что я не поставил временную зону в вызовах MockChronometer и TimeUtils — то тогда вот мой вам респект. Оставлять их в тестах на откуп текущей временной зоне — значит сделать тест хрупким. На самом же деле оба класса по-умолчанию оперируют в временной зоне UTC всегда, когда зона не указана специально в аргументах методов.  
  
В Java 8 в новом Date Time API появился интерфейс [java.time.Clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html), который введен ровно для тех же целей — но я не уверен, что общественность это уже оценила.   
  
Альтернативный подход — запускать тесты в отдельной [JVM с указанием агента](http://habrahabr.ru/post/230239/), который будет производить инструментацию кода с целью перехвата вызовов к System.nanoTime() и System.currentTimeMillis(). Такой подход я не пробовал, а беглый поиск готовых решений [не](http://stackoverflow.com/questions/17398279/is-it-possible-to-freeze-system-currenttimemillis-for-testing) [предлагает](http://stackoverflow.com/questions/2001671/override-java-system-currenttimemillis-for-testing-time-sensitive-code). Более здравым вариантом кажется простой препроцессинг исходного кода в процессе сборки — замена вызовов System.nanoTime(), System.currentTimeMillis(), new Date(), Calendar.getInstance() на вызовы к своему классу.

Spring Framework

8 лет назад в Spring был LocaleResolver, но не было TimezoneResolver (что, как мне кажется, вполне характеризует общее отношение к проблеме). Пришлось написать свой комплект, а заодно сделать сабкласс DispatcherServlet.  
  
После не очень многочисленных, но достаточно настойчивых [просьб сообщества](https://jira.spring.io/browse/SPR-1528) (моих в том числе), [штатный резолвер временной зоны](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/i18n/TimeZoneAwareLocaleContext.html) запроса был введен в 4-й версии фреймворка. 

MVC

Отдельный вопрос — как, зная временную зону в контроллере, правильно установить ее для шаблонизатора.  
  
В FreeMarker предусмотрена специальная [настройка](http://freemarker.incubator.apache.org/docs/ref_directive_setting.html) для текущего рендеринга:

<#setting time\_zone="Europe/Moscow">

Для JSP также можно указать временную зоны индивидуально для [одиночного вывода](http://www.tutorialspoint.com/jsp/jstl_format_formatdate_tag.htm) или сразу для [всего блока](http://www.tutorialspoint.com/jsp/jstl_format_settimezone_tag.htm):

<fmt:formatDate type="both" value="${now}" timeZone="Europe/Moscow"/>

<fmt:timeZone value="Europe/Moscow">

<fmt:formatDate type="both" value="${now}"/>

</fmt:timeZone>

В Velocity тоже [что-то есть](https://velocity.apache.org/tools/devel/javadoc/org/apache/velocity/tools/generic/DateTool.html#format%28java.lang.String,%20java.lang.Object,%20java.util.Locale,%20java.util.TimeZone%29), но я лично не пробовал.

Хранение времени в БД

Самый бронебойный способ хранения временной точки в БД — эта передача значения java.util.Date#getTime() в базу в виде простого числового long-значения Unix-time. Соответственно при чтении мы преобразуем long в java.util.Date с помощью конструктора. Это можно сделать в конверторах Hibernate или RowMapper'ах. База данных в этом случае ничего не знает про время, поэтому никаких внезапных эффектов мы получить не сможем. Если очень надо вывести временную точку в виде метки, то, например в MySQL, всегда можно вызвать метод FROM\_UNIXTIME.  
  
Такой способ подходит, если в запросах в БД и/или в хранимых процедурах нет операций с временем. Если же такие операции есть (чего, для простоты разработки, лучше конечно бы избегать), то вы уже в курсе, что без временной зоны они не проходят. В этом случае, надо понять какая именно временная зона действует в ходе операции преобразования или ввода/вывода:

* временная зона указанная по-умолчанию для сервера;
* временная зона указанная по-умолчанию для СУБД;
* временная зона указанная по-умолчанию для базы данных;
* временная зона указанная по-умолчанию для таблицы;
* временная зона указанная по-умолчанию для соединения;
* временная зона хранящаяся в ячейке вместе с временной меткой.

Я ни в коем случае не хочу сказать, что не надо хранить время в специально предназначенных для этого типах. Просто вариант с прямым хранением long очень сложно как-либо сломать (я не смог придумать как) и такой способ никак не зависит от типа хранилища.  
  
При хранении времени в специально предназначенных для этого типах нужно обязательно проверить, что при указанных настройках БД и ее драйвера указанное значение времени пишется, читается и выводится в родной консоли управления этой БД консистентно даже при смене часовых поясов на ходу.   
  
Неконсистентность может быть вызвана тем, что БД на самом деле может хранить временную метку, а не точку. Например в случае MySQL для хранения времени существуют два стандартных типа: TIMESTAMP and DATETIME.   
  
Судя по официальной документации, TIMESTAMP хранит именно временную точку и, получив значение «2015-01-01 12:00:00 MSK» от клиента в московской временной зоне, вернет «2015-01-01 09:00:00 UTC» для другого клиента во временной зоне UTC, что соответствует одной временной точке и совершенно правильно по своей сути. А с типом DATETIME, получив от MSK-клиента «2015-01-01 12:00:00 MSK», сервер MySQL вернет UTC-клиенту «2015-01-01 12:00:00 UTC», что соответствует уже другой временной точке и все дальнейшие расчеты будут неверными.   
  
Проверим MySQL. Сначала все подготовим:

Запуск БД

Устанавливаем в сессии зону «Europe/Moscow» и создаем запись:

Europe/Moscow

Меняем временную зону сессии на «UTC» и читаем запись снова:

UTC

Видим, что временная метка поля t1 изменилась — как и должно быть, а временная метка для поля t2 при смене временной зоны не изменилась и теперь соответствует другой точке на числовой оси.   
  
Для большинства БД существуют специфичные типы, которые хранят только дату и только время. Такие типы невозможно привести к временной точке, не указав временную зону (о чем мы уже знаем), но также необходимо дополнить эти значения недостающей частью (временем или датой соответственно). На самом деле, можно и не приводить такие типы к временным точкам, если в этом нет никакой необходимости — например в случае простого вывода дня исторического события. Просто нужно помнить, что это метка (или даже ее часть), а не временная точка на оси.   
  
Многие БД имеют разновидности типов, которые кроме самой временной точки хранят дополнительную и информацию о временной зоне в которой она была введена. Например, это может быть полезно, если нужно знать какая именно временная зона была первична для этого значения.

NTP

Вне зависимости от причин, по которым локальное время в системе отклоняется от реального, на помощь приходят сервисы поставки точного времени с различными протоколами, самый популярный из которых — это протокол [NTP](https://ru.wikipedia.org/wiki/NTP).   
  
Атомные часы — достаточно дорогое удовольствие, поэтому чтобы не перегрузить связанные с ними сервера, в NTP выстраивается [целая иерархия](http://habrahabr.ru/post/79461/) для обслуживания запросов рядовых пользователей.  
  
Клиент регулярно опрашивает сразу несколько сервисов точного времени в разных подсетях одновременно, [производит компенсацию](https://en.wikipedia.org/wiki/Network_Time_Protocol#Clock_synchronization_algorithm) времени потраченного на отправку и прием UDP датаграмм, откидывает значения являющиеся выбросами и затем усредняет их [специальным алгоритмом](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9C%D0%B0%D1%80%D0%B7%D1%83%D0%BB%D0%BB%D0%BE).  
  
[Заявляется](https://ru.wikipedia.org/wiki/NTP), что при использовании в публичных сетях возможная точность синхронизации может достигать 10мс.

Уведомления

В большинстве сервисов мобильные уведомления (пуши и СМС) можно поделить на два класса: срочные (сообщения, новые дружбы, платежи) и несрочные (промо, реклама, предложения, изменения регламента). Первые пользователь скорее всего ожидает получить сразу же при наступлении события, для вторых же события как такового может и не быть, и пользователя лучше беспокоить в комфортное для него время. Комфортным временем по-умолчанию можно считать например период с 10 утра до 20 вечера. Если это критично, комфортное время можно позволить указывать вручную индивидуально.  
  
Так или иначе, этот период мы обязаны трактовать в какой-то временной зоне и вполне очевидно, что это будет временная зона пользователя. В принципе, если сервис работает только для какой-то ограниченной территории (например для одного города) все это можно не учитывать и считать, что временная зона всех пользователей совпадает с временной зоной сервера, но для распределенных сервисов этого недостаточно.  
  
Поэтому временную зону необходимо сохранять для фоновых задач — как минимум индивидуально для пользователя, а еще лучше отдельно для каждого устройства пользователя. В современных реалиях одна и та же учетная запись приложения может использоваться сразу на нескольких устройствах: например на стационарном Android TV, который всегда находится дома в Москве; и на Android-планшете, которые уезжает вместе с пользователем в отпуск в Таиланд. Поэтому возможно, что уведомления на оба устройства придется отсылать в разное время. Имея временную зону пользователя и комфортное для него время, всегда можно рассчитать точку на временной оси, когда мы можем начать его спамить. 

Пример

A.M. / P.M.

Я предполагаю, что тут нет людей, которые бы уже не поняли, что речь идет о [12-часовом представлении](https://ru.wikipedia.org/wiki/12-%D1%87%D0%B0%D1%81%D0%BE%D0%B2%D0%BE%D0%B9_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8) временных меток.   
  
Но не только лишь все знают, как именно переводятся эти аббревиатуры: a.m. (лат. ante meridiem дословно — «до полудня») и p.m. (лат. post meridiem дословно — «после полудня»).  
  
А сюрпризом для многих будет то, что полночь — это не 12pm и даже не 0am, а очень даже 12am. Аналогично, полдень — это не 12am и не 0pm, а 12pm. В таких обозначениях путаются даже жители привычных к такому формату стран, поэтому придумывают [различные трюки](https://ru.wikipedia.org/wiki/12-%D1%87%D0%B0%D1%81%D0%BE%D0%B2%D0%BE%D0%B9_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8#.D0.9F.D1.80.D0.BE.D0.B1.D0.BB.D0.B5.D0.BC.D1.8B_.D0.B2_.D0.BE.D0.B1.D0.BE.D0.B7.D0.BD.D0.B0.D1.87.D0.B5.D0.BD.D0.B8.D1.8F.D1.85_.D0.BF.D0.BE.D0.BB.D1.83.D0.B4.D0.BD.D1.8F_.D0.B8_.D0.BF.D0.BE.D0.BB.D1.83.D0.BD.D0.BE.D1.87.D0.B8). 

Часы реального времени

На любой железке с любой архитектурой есть специальный чип с подключенной батарейкой — это [часы реального времени](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%8B_%D1%80%D0%B5%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8) или в терминологии Linux «The Hardware Clock».  
  
  
  
Они бесшумно тикают своими электронами вне зависимости от внешних условий: подано ли основное питание на системный блок, установлена ли операционная система, а также от наличия или отсутствия каких-либо ваших действий. Их ход прерывается только после того как энергия в батарейке иссякнет, что на самом деле произойдет с вашим оборудованием [достаточно нескоро](https://www.maximintegrated.com/en/design/tools/calculators/product-design/battery.cfm).  
  
Для управления часами Hardware Clock (фактически для отправки команд в этот чип) в Linux существует специальная команда hwclock. Поскольку она напрямую общается с оборудованием (как правило это /dev/rtc), выполнять ее нужно с правами root.

$ sudo hwclock

Wed 30 Dec 2015 17:59:12 MSK -0.328637 seconds

Команда автоматически приводит время полученное из Hardware Clock к временной зоне системы. Но что это за странное смещение в конце вывода?  
  
Дело в том, что на самом деле в ядре Linux есть еще другие, свои собственные часы (The System Time). Ядро Linux считывает показания с чипа один раз, в начале загрузки ядра, после чего тикает уже само по себе в ходе прерываний. Скорее всего это сделано по одной простой причине — читать данные с железного чипа Hardware Clock по последовательной шине на каждый пользовательский запрос относительно долго, поэтому проще вести свой локальный, исключительно программный счетчик. После того как ядро Linux прочитало значение с Hardware Clock, с последними теоретически можно производить любые операции — на время в приложениях это никак не повлияет.   
  
Таким образом в системе параллельно тикают двое часов — настоящие электронные часы и программные часы в ядре. Между ними неизбежно возникает разница, которая и отображена в выводе команды hwclock  
  
Есть два варианта решения этой проблемы (а для кого-то это и вовсе не проблема).   
  
Рассмотрим первый вариант, предположив также что система не подключена к сети. Поскольку железные часы Hardware Clock работают круглыми сутками без каких либо перерывов — то и ошибаются они в среднем на одну и ту же величину каждый день. Утилита hwclock может нивелировать это через специальный механизм коррекции при чтении значения из чипа. Пользователю достаточно два раза установить Hardware Clock с каким-то существенным промежутком времени между установками. Утилита hwclock сама посчитает на сколько именно ошибаются Hardware Clock в течении суток, после чего сохранит эту величину в файле /etc/adjtime. После этого мы можем периодически читать значение из Hardware Clock и устанавливать значение System Time по нему, при этом утилита сама произведет коррекцию накопленной ошибки на сохраненную ранее величину дневной ошибки.  
  
Второй вариант предполагает, что у нас есть какой-то способ периодической правильной установки System Time в ядре. Скорее всего, это какой-то внешний источник точного времени, с помощью которого мы можем приводить System Time в актуальное состояние. Все что остается после этого — попросить ядро периодически (раз в 11 секунд) скидывать верное значение ядра System Time в чип Hardware Time.  
  
Подробнее обо всем этом можно прочитать в [мануале команды hwclock](http://linux.die.net/man/8/hwclock).  
  
Также наверное стоит рассказать о том, что система Linux может интерпретировать значения из Hardware Clock двумя способами. Связано это с тем, что Hardware Clock хранит время в [виде счетчиков yyyy, MM, dd, HH, mm, ss](https://wiki.archlinux.org/index.php/Time#Hardware_clock_and_system_clock). А как было уже сказано выше, без временной зоны эти счетчики нельзя привязать к точке на временной оси.   
  
На самом деле тут всего два варианта: это будет или UTC, или локальная временная зона (та, что установлена в операционной системе по-умолчанию).  
  
Для начала зайдите в BIOS и посмотрите на часы в главном меню. Сравните показания часов в BIOS и своих наручных часов, если они совпадают, то у вас часы BIOS идут по локальному времени (поздравляю — скорее всего у вас Windows компьютера); а если не совпадают, то часы BIOS установлены во временной зоне UTC (или другой вариант — они идут неправильно).  
  
Windows по умолчанию требует, чтобы время в чипе Hardware Clock соответствовало локальному времени. А Linux легко может работать как с локальным временем в Hardware Clock, так и временем по UTC (последнее предпочтительнее). Поэтому, как правило, при двойной загрузке системы часы в BIOS идут по локальному времени специально для Windows, а Linux к этому приспосабливается.   
  
Посмотреть текущий режим в Debian/Ubuntu можно так:

$ cat /etc/default/rcS | grep UTC

*# assume that the BIOS clock is set to UTC time (recommended)*

UTC=yes

Хаки времени

Если нужно заставить какую-то программу думать, что в данный момент для нее время отличается от времени всей системы, то в репозиториях Ubuntu/Debian уже есть [утилита faketime](http://manpages.ubuntu.com/manpages/wily/man1/faketime.1.html), которая перехватывает и модифицирует системные вызовы.

$ date

Wed Dec 30 22:53:11 MSK 2015

$ faketime -f '+2y' date

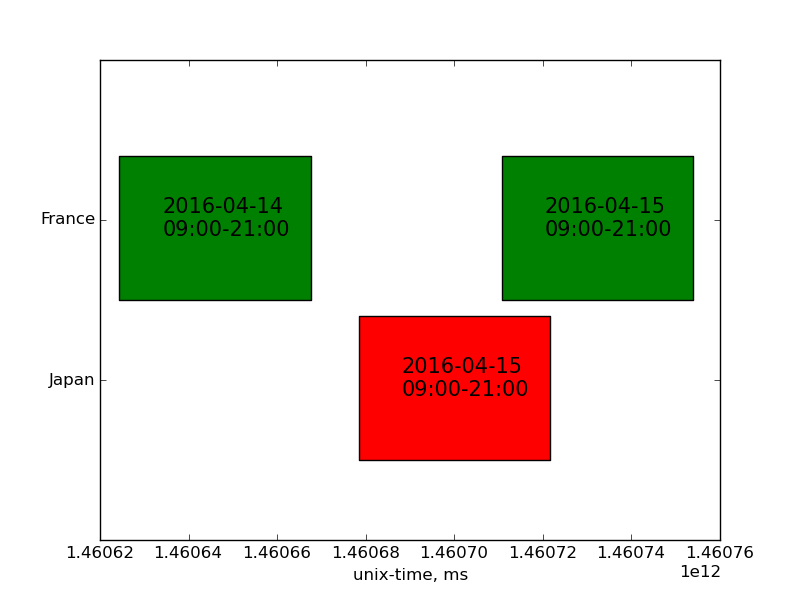
Fri Dec 29 22:53:39 MSK 2017

День рождения

Если кто-то скажет, что родился «15 апреля», мы получим возможность поздравлять этого человека с днем рождения каждый год. Если он скажет, что родился «15 апреля 2001 года», мы получим возможность узнать еще и его возраст. Но и в том, и другом случае это никак не соответствует никакой временной точке на оси. Во-первых, не указано время рождения. Во-вторых, не указана временная зона рождения. Хотя, теоретически, временную зону даты рождения можно узнать, если указано точное место рождения.  
  
Как именно нам рассчитывать временную точку в которую мы можем отправить поздравления самому пользователю, а в какую временную точку отправлять уведомления о его дне рождения его друзьям? Как вариант, можно предложить следующее:

* Отправка нотификации пользователю — t(u). Поскольку временная зона рождения нам неизвестна, используем текущую временную зону пользователя. Поскольку время рождения неизвестно, используем комфортное для пользователя время.
* Отправка нотификации другу — t(f). Также комбинируем дату рождения пользователя, временную зону друга и комфортное время друга.

Japan-France

  
  
Время t(f) может быть сильно больше чем время t(u) — например, если пользователь находится в Японии, а его друг — в Европе. В этом случае получается, что к моменту, когда другу в Европе рано утром придет уведомление, сам пользователь в Японии уже возможно закончит праздновать свое ДР. В таком случае можно сдвинуть время уведомления друга на день назад, а во фразу уведомления добавить слово «завтра».

Пограничный возраст

Есть еще один, крайне интересный для меня момент, уже юридического характера. Есть целый ряд граничных возрастов: возраст совершеннолетия, возраст сексуального согласия, возраст уголовной ответственности.  
  
Предположим, что некто родился в 2000.01.10 00:00:01 во Владивостоке. Совершил нетяжкое преступление 2016.01.09 23:59:59 в Москве. По записи дня рождения в паспорте (2000.01.10) и дня преступления в протоколе (2016.01.09) получается что 16 лет человеку еще не исполнилось. Однако 2016.01.09 23:59:59 в Москве — это уже будет 2016.01.10 во Владивостоке (где он родился) и тогда 16 лет ему уже есть. В обратной ситуации, человеку родившемуся в Москве и совершившему преступление во Владивостоке уже исполнится 16, а вот если посчитать по московскому времени — получается что еще нет.  
  
Для устранения этого казуса в судебной практике [используется норма](http://citizen.joosy.ru/questions/240219-%D0%9A%D0%BE%D0%B3%D0%B4%D0%B0-%D1%8E%D1%80%D0%B8%D0%B4%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D0%B5%D1%82%D1%81%D1%8F-%D1%81%D0%BE%D0%B2%D0%B5%D1%80%D1%88%D0%B5%D0%BD%D0%BD%D0%BE%D0%BB%D0%B5%D1%82%D0%B8%D0%B5--%D0%B2-%D0%B4%D0%B5%D0%BD%D1%8C-%D1%80%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F-%D0%B8%D0%BB%D0%B8-%D0%BD%D0%B0-%D1%81%D0%BB%D0%B5%D0%B4%D1%83%D1%8E%D1%89%D0%B8%D0%B9-%D0%B4%D0%B5%D0%BD%D1%8C-), при которой активация прав/ответственности наступает с 00:00:00 в день, следующий за ожидаемым. То есть уголовная ответственность наступит в 2016.01.11 00:00:00 по месту совершения события — в этом случае человеку точно будет 16 лет, где бы он не находился.

Выводы

* Не игнорируйте таймзону
* Если у вас почти готова машина времени, то я не рекомендую использовать стандартную библиотеку Java для точного расчета смещения в прошлое. Точные путешествия в будущее по временным меткам невозможны в принципе.
* Также текущая версия JDK не подходит для написания внутренних систем планирования на космических кораблях Галактической империи.

[Вторая часть статьи](http://habrahabr.ru/post/274905/) — про новое Date Time API в Java 8.