

The
Pragmatic
Programmers

Программирование на Эрланге

Программное
обеспечение для
параллельного мира



Джо Армстронг

Программирование на Эрланге

Наш мир существует во времени параллельно и одновременно.

Если мы хотим писать программы, которые ведут себя также как объекты реального

мира, то эти программы должны иметь параллельную структуру.

Используйте для этого язык, который был специально разработан для написания параллельных приложений и их разработка станет для вас гораздо проще.

Модель программирования Эрланга — это то как мы, на самом деле, думаем и взаимодействуем.

Джо Армстронг

Глава 1. Начало

О, нет! Еще один язык программирования! Зачем мне его учить? Разве уже существующих не достаточно?

Я понимаю такую вашу реакцию. Сегодня в мире уже существует множество языков программирования, зачем же изучать еще один?

Вот вам пять причин, почему вам может быть полезно изучение Эрланга:

- Вы хотите писать программы которые будут работать быстрее, если их запустить на много-ядерных компьютерах.
- Вы хотите писать отказоустойчивые приложения, которые могут быть модифицированы прямо в процессе их работы.
- Вы что-то слышали о «функциональном программировании» и вас заинтересовало, насколько оно полезно на практике.
- Вы хотите использовать язык который был многократно проверен в «боевых» условиях реального мира в масштабных промышленных продуктах и который имеет громадные библиотеки и активное сообщество пользователей.
- Вы не хотите стачивать ваши пальцы от набивания множества строчек кода.

Разве все это возможно? В разделе 20.3 *Работа с SMP Эрлангом* мы рассмотрим некоторые программы, которые линейно ускоряют свою работу на 32-ядерном компьютере. В главе 18 *Разработка систем с помощью OTP*, мы рассмотрим как создавать высоко-надежные системы, которые работают круглосуточно, без остановки, в течении многих лет. В разделе 16.1 *Путь к обобщенному (Generic) серверу*, мы поговорим о написании серверов, приложения на которых могут быть модифицированы «на лету», без остановки работы сервера.

Во множестве мест этой книги, мы будем восхвалять добродетели функционального программирования, которое, в частности, запрещает наличие у программ побочных эффектов. Побочные эффекты и параллельное программирование просто

несовместимы. Либо вы программируете с побочными эффектами последовательные программы, либо параллельные, но без побочных эффектов. Выбор за вами, но третьего, просто, не дано.

Эрланг — это язык в котором параллельность программ встроена в сам язык, а не связана с операционной системой. Эрланг облегчает создание параллельных программ, путем моделирования окружающего мира в виде множества параллельных процессов, которые взаимодействуют, только обмениваясь сообщениями. В мире Эрланга все параллельные процессы существуют без взаимных блокировок, методов синхронизации и возможности воздействия на общую память, так как ее просто не существует.

Программы на Эрланге могут состоять из тысяч и миллионов очень «маленьких» процессов, которые могут исполняться на одном процессоре компьютера, на многоядерном процессоре, или же на сети компьютеров.

1.1. Краткое описание глав книги

- Глава 2 «Введение» - это, типа, быстро «запрыгнуть и осмотреться» что это все такое.
- Глава 3 «Последовательное программирование» - это первая из двух глав о последовательном программировании на Эрланге. В ней, например, вводятся такие понятия, как «соответствие по образцу» и «неразрушающее присвоение».
- Глава 4 «Исключения» посвящена обработке исключений. Не бывает программ без ошибок. И эта глава посвящена обнаружению и обработке ошибок в последовательных программах на Эрланге.
- Глава 5 «Продвинутое последовательное программирование» - это вторая глава о последовательном программировании на Эрланге. Она рассматривает несколько тем по-сложнее и заканчивает рассмотрение последовательного программирования.
- Глава 6 «Компиляция и запуск программ» рассказывает о различных способах компиляции и запуска ваших программ.
- В главе 7 «Параллельность» мы делаем пересадку. Это не-техническая глава. Он посвящена, скорее, идеологии того, как мы программируем вообще и как мы видим наш окружающий мир.
- Глава 8 «Параллельное программирование» посвящена параллельности в Эрланге. Как нам создать параллельные процессы? Как они будут взаимодействовать между собой? Насколько затратно по времени их создание?
- Глава 9 «Ошибки в параллельных программах» рассматривает эту тему. Что произойдет при падении процесса? Как мы можем обнаружить его падение и что мы можем с этим сделать?

- Глава 10 «Распределенное программирование» является введением в эту тему. В ней мы рассмотрим несколько небольших распределенных программ и покажем как их запускать на кластере узлов Эрланга или на независимых компьютерах в сети, используя механизмы socket-взаимодействия.
- Глава 11 «Маленький IRC» посвящена одному этому приложению. Мы сложим вместе темы параллельности и socket-распределенности в нашем первом, нетривиальном, приложении: небольших IRC-подобных клиенте и сервере.
- Глава 12 «Интерфейсы взаимодействия» посвящена вопросам взаимодействия программ на Эрланге и программ на других языках.
- Глава 13 «Программирование с Файлами» дает множество примеров использования файлов в программах.
- Глава 14 «Программирование с Socket-ами» дает примеры таких программ. Мы рассмотрим как построить параллельные и последовательные серверы на Эрланге. В завершении этой главы мы рассмотрим второе масштабное приложение: SHOUTcast медиа-сервер. Который может раздавать MP3 данные используя SHOUTcast протокол.
- Глава 15 «ETS и DETS: механизмы хранения больших объемов данных» описывает низко-уровневые модули Эрланга ets и dets. Модуль ets предназначен для очень быстрых, «деструктивных» операций с хеш-таблицами и их данными в памяти, а dets разработан для хранения данных на диске.
- Глава 16 «Введение в OTP» именно этому и посвящена. OTP — это набор библиотек и операционных процедур Эрланга предназначенных для построения серьезных промышленных приложений. В этой главе вводится понятие поведения (behavior) – центральной концепции OTP. Используя типы поведения мы можем сосредоточиться на только на функционале компонент наших приложений, в то время как поведенческое окружение OTP позаботится обо всем остальном. Это окружение, например, может само реализовать отказоустойчивость или масштабируемость нашего приложения, в то время как (написанные нами) модули обратного вызова приложения реализуют всю его специфическую функциональность. Эта глава начинается с общего обсуждения как работают поведения вообще, а потом переходит к описанию поведения gen_server (обобщенный сервер), как части стандартной библиотеки Эрланг OTP.
- Глава 17 «Mnesia («Мнезия») - база данных Эрланга» рассказывает о базе данных (СУБД) Mnesia. Мнезия — это интегрированная в Эрланг СУБД с очень быстрым временем отклика в реальном времени. Она может быть сконфигурирована на репликацию данных по нескольким, физически различным, узлам для обеспечения отказоустойчивости.
- Глава 18 "Создание систем при помощи OTP" - это следующая глава об OTP. Она посвящена практическим вопросам создания OTP-приложений. Реальные приложения нуждаются для работы во множестве мелких, необходимых деталей. Они должны запускаться и останавливаться определенным образом. Если они или их компоненты падают, они должны быть определенным образом перезапущены. Также нужен журнал ошибок, позволяющий определить, что же произошло при

падении. Эта глава как раз и описывает все эти мелочи, позволяющие создавать полноценные ОТП приложения.

- Глава 19 "Мультиядерная прелюдия" - это краткое объяснение, почему Эрланг хорошо приспособлен для программирования многоядерных компьютеров. Мы обсудим, в общем, проблему разделяемой памяти и параллельных программ с передачей сообщений и почему мы свято верим, что параллельные языки без взаимного влияния идеально подходят для программирования многоядерных компьютеров.
- Глава 20 "Программирование многоядерных ЦПУ" посвящена программированию многоядерных компьютеров. Мы поговорим о том, как сделать так, чтобы ваша программа эффективно использовала многоядерность компьютера. Мы представим вам несколько абстрактных понятий используемых для ускорения работы последовательных программ на многоядерных компьютерах. Кроме того мы проведем ряд измерений и напишем нашу третью серьезную программу - движок полнотекстового поиска. Для его реализации мы, для начала, напишем функцию mapreduce - высоконивневую функцию, используемую для распараллеливания вычислений на некоторое множество вычислительных элементов.
- Приложение A - описывает систему типов применяемых при документировании функций в Эрланге.
- Приложение B - описывает установку Эрланга в операционной системе Windows (и как сконфигурировать Emacs, во всех операционных системах).
- Приложение C - содержит каталог ресурсов по Эрланг.
- Приложение D - описывает библиотеку lib_chan предназначенную для создания socket-распределенных приложений.
- Приложение E - рассматривает техники анализа профилирования, отладки и трассировки вашего кода.
- Приложение F - содержит односторонние описания самых используемых стандартных модулей Эрланга.

1.2. Еще раз с начала

Однажды программисту попалась в руки книга, описывающая забавный язык. У него был незнакомый синтаксис, равенство вовсе не означало равенства, переменным не разрешалось изменяться. Хуже того, он даже не был объектно-ориентированным. А программы были, как бы это сказать, немного другими...

Но не только программы были другими, но и весь подход к программированию был другим. Автор все время говорил про параллельность и распределенность программ, про их отказоустойчивость и про метод программирования, называемый параллельно-ориентированное программирование - что бы это там не значило.

Но некоторые примеры были весьма забавными. В тот вечер, программист рассматривал пример программы для чатов. Он был очень маленьким и легким для понимания, даже не смотря на немного странный синтаксис. Невозможно, чтобы все было так просто.

Основная программа была простой, а с помощью еще нескольких строчек кода появились и возможности обмена файлами и зашифрованные разговоры. Программист начал нажимать клавиши на клавиатуре...

О чем, вообще, эта книга?

Она о параллельности. Она о распределённости. Она об отказоустойчивости. Она о функциональном программировании. Она о написании распределенных параллельных систем без взаимных блокировок и общих областей, а только на чистом обмене сообщениями. Она о ускорении ваших программ на многоядерных процессорах. Она о написании распределенных приложений, которые позволяют людям взаимодействовать друг с другом. Она о методах дизайна и поведения систем для написания отказоустойчивых и распределенных приложений. Она о моделировании параллельности мира и отображении этих моделей в компьютерные программы. Этот процесс я называю параллельно-ориентированным программированием.

Мне было очень приятно писать эту книгу. Я надеюсь, что вам будет также приятно ее читать.

А теперь, начните читать книгу, писать примеры и получать от этого удовольствие.

1.3. Благодарности

Многие люди помогли мне в подготовке этой книги и я хотел бы всех их сердечно поблагодарить здесь.

Во-первых, Дейва Томаса, моего редактора: Дейв учил меня как писать и засыпал невероятным количеством бесконечных вопросов. Почему это так? А почему это эдак? Когда я начал писать книгу, Дейв сказал мне что я пишу ее в стиле "проповедей с высокой скалы". Он сказал мне: "я хочу, чтобы ты просто говорил с людьми, а не проповедовал им". Так книга стала гораздо лучше, спасибо, Дейв.

Далее, у меня был небольшой комитет моей поддержки, состоящий из экспертов по языку. Они помогли мне решить, что оставить за бортом рассмотрения. А также, помогли мне прояснить пару деталей, тяжелых для рассмотрения. Спасибо (без определенного порядка) Бьерну Густавсону, Роберту Вердингу, Костису Сагонас, Кеннет Лундин, Ричарду Карлосону и Ульфу Вайгеру.

Спасибо также Клаису Викстрёму который дал ряд полезных советов по Мнезии, Рикарду Грину за SMP Эрланг, и Хансу Нильсону за морфологический алгоритм, использованный в индексировании текстов.

Шон Хинди и Ульф Вайгер помогли мне понять, как использовать многие внутренние аспекты ОТР, а Сергей Алейников объяснил мне активные сокеты, так что даже я смог их понять.

Хелен Тейлор (моя жена) критически прочитала несколько глав этой книги и обеспечила меня многими сотнями чашек чая в подходящие для этого моменты. Более того, она как-топравлялась с моим, скорее, одержимым поведением последние семь месяцев. Спасибо, также, Томасу и Клэр, и еще спасибо Баху, Генделю, Зорро, Дейзи и Доррис, которые помогали мне не сойти с ума, мурлыкали, когда их чесали и помогали мне добраться домой по правильному адресу.

И, наконец, спасибо всем читателям бета-версии этой книги, которые прислали мне свои замечания. Я вас проклинал и я безмерно благодарен вам. Когда была опубликована первая бета этой книги, я был просто не готов, что она будет прочитана за два дня и просто распотрошена по кусочкам, каждая страница, вашими комментариями. Но это привело к созданию, гораздо более улучшенной версии книги, чем я вообще мог себе представить. Когда (как это было несколько раз) десятки людей писало "я не понимаю, что написано на этой странице", я был вынужден обдумать все снова и переписать указанный материал. Люди, спасибо вам всем, за вашу помощь.

Джо Армстронг

Май 2007-го года.

Глава 2. Приступаем к изучению

2.1. Введение

Как и с любым другим языком программирования, вы пройдете через несколько стадий на вашем пути к мастерству в Эрланге. Давайте посмотрим на эти стадии, которые мы охватываем в данной книге и на то что вы изучите по мере своего продвижения.

Стадия 1: Я не уверен...

Когда вы новичок, вам надо научиться, с начала, запускать систему, выполнять

команды в оболочке Эрланга, компилировать простые программы и, вообще, познакомиться с этим языком. (Эрланг - это маленький язык, так что это не займет много времени.)

Давайте разделим это на более мелкие куски. Как новичок, вы сделаете следующее

- Убедитесь, что на вашем компьютере установлена работающая система Эрланг.
- Научитесь запускать и останавливать командную оболочку Эрланга.
- Узнаете как набирать и выполнять различные выражения Эрланга в его командной оболочке, а также понимать результаты такого выполнения.
- Увидите как создавать и модифицировать программы на Эрланге с помощью вашего любимого текстового редактора.
- Поэкспериментируйте с компиляцией и выполнением ваших программ в командной оболочке Эрланг

Стадия 2: Мне комфортно с Эрлангом

Итак, вы уже немного научились работать с языком Эрланг. Поскольку вы уже познакомились с этим языком, то вы готовы изучать Главу 5. Углубленное последовательное программирование.

На этой стадии вы окончательно познакомитесь с Эрлангом и мы сможем перейти к его более интересным темам:

Вы узнаете о более хитрых техниках использования оболочки Эрланга. Оболочка может делать гораздо больше, чем мы будем себе представлять после первого знакомства с ней. (Например, вы можете в ней вызывать заново и редактировать ваши прошлые выражения и команды. Об этом рассказывается в разделе 6.5
*Редактирование команд в оболочке Эрланга.)

Вы начнете изучение библиотек (называемых в Эрланге модулями). Большинство программ, из числа тех что я написал, могут быть написаны с использованием всего пяти модулей: lists, io, file, dict и gen_tcp . Следовательно мы будем активно пользоваться этими модулями на протяжении всей книги.

По мере того, как ваши программы будут становиться все больше, вам потребуется знать, как автоматизировать их компиляцию и запуск. Наилучшим решением для этого является утилита make. Мы научимся как можно контролировать этот процесс с помощью написания make-файлов. Об этом рассказывается в Разделе 6.4
Автоматизация компиляции с помощью Make-файлов.

В большом мире программирования на Эрланге активно используется большая коллекция библиотек, называемая OTP (от ее названия - Open Telecom Platform -

Открытая Платформа для телекоммуникационных приложений). По мере накопления вами опыта работы с Эрлангом, вы начнете понимать, что владение OTP сбережет вам множество времени и сил при написании серьезных приложений. В конце-концов, зачем заново разрабатывать колесо, когда кто-то уже реализовал ту функциональность, которая вам нужна? Мы изучим основу OTP - *поведения*, в частности `gen_server`.

Одно из основных применений Эрланг - это написание распределенных программ, так что теперь настало время поэкспериментировать с этим. Начать можно с примеров приведенных в Главе 10 *Распределенное программирование*, а потом эту тему можно расширить насколько вы это пожелаете.

Стадия 2.5: Я могу изучить дополнительные темы

Вам не надо изучать каждую главу этой книги при первом ее прочтении.

В отличии от большинства других языков, с которыми вы встречались ранее, Эрланг это параллельный язык программирования, и это делает его особенно удобным для написания распределенных программ, а также для программирования современных многоядерных и SMP (Symmetric multiprocessing - Симметрично-мультипроцессорных) компьютеров. Множество Эрланг программ начинают просто работать быстрее, будучи запущенными на многоядерных или на SMP машинах.

Программирование на Эрланге основывается на новой парадигме программирования, которую я называю *параллельно-ориентированное программирование (ПОП, POP - parallel-oriented programming).

Когда вы используете ПОП, вы разбиваете проблему на множество мелких процессов и определяете естественный параллелизм в ее решении. Это важнейший шаг при написании любой параллельной программы.

Стадия 3: Я - Эрланг Мастер

Теперь вы являетесь мастером в языке можете написать полезные распределенные программы. Но чтобы достичь истинного мастерства вы должны изучить еще больше:

- *Mnesia*. Дистрибутив Эрланга поставляется всем бесплатно вместе с встроенной быстрой, реплицируемой базой данных называемой *Mnesia*. Она была изначально разработана для телекоммуникационных приложений где производительность и отказоустойчивость являются ключевыми критериями. Сейчас она широко используется и в различных не-телекоммуникационных приложениях.
- Интерфейсы с программами написанными на других языках и использование *встроенных драйверов*. Это рассматривается в разделе 12.4 *Встроенные драйверы*.

- Свободное использование основанных на поведении деревьев супервизоров, сценариев старта и так далее. Об этом рассказывается в Главе 18 *Разработка систем при помощи OTP*.
- Как запустить и оптимизировать вашу программу для многоядерных компьютеров. Об этом рассказывается в Главе 20 *Программирование многоядерных процессоров*.

Самый главный урок

Есть одно правило которое вы должны помнить на протяжении всей этой книги: программирование - это здорово, приятно, весело и интересно. И лично я считаю, что программирование распределенных приложений, таких как программа чата или обмена мгновенными сообщениями - это гораздо более приятно и весело, чем программирование обычных последовательных приложений. То что вы можете сделать на одном компьютере - ограничено его возможностями, но возможности сети компьютеров - практически безграничны. И Эрланг обеспечивает вам идеальную среду для экспериментов с сетевыми приложениями и для построения промышленных систем.

Чтобы помочь вам освоиться со всем этим я перемешал главы посвященные приложениям реального мира с техническими главами по системе Эрланг. Вы должны рассматривать эти приложения как отправные точки для ваших собственных экспериментов. Возьмите их, доработайте их и используйте их так, как я и предположить того не мог и я буду от этого очень счастлив.

2.2. Инсталляция Эрланга

Прежде чем вы сможете что-либо сделать, вам надо убедиться, что у вас установлена работающая версия Эрланг на вашей системе. Идите в командную консоль и наберите там erl:

```
$ erl1
Erlang (BEAM) emulator version 5.5.2 [source] ... [kernel-poll:false]
Eshell V5.5.2 (abort with ^G)
1>
```

В Windows erl из консоли сработает только если он установлен и переменная окружения PATH указывает на его исполняемый файл тоже. Если вы установили Эрланг в Windows стандартным путем, то вы сможете его вызвать через меню Старт > Все Программы > Erlang OTP . В приложении В я рассказываю как я настроил Эрланг в Windows на совместную работу с

Примечание: В этой книге я буду показывать приветственное обращение Эрланга (приведенное чуть выше) только изредка. Эта информация полезна только если вы хотите сообщить об ошибке. Я ее показал тут только для того, чтобы вы не волновались, увидев подобное. В большинстве примеров я его показывать не буду, если, конечно, это не будет необходимо.

Если вы увидели приветствие его командной оболочки, значит Эрланг установлен на вашем компьютере. Выходите из него - нажмите **Ctrl+G** а потом еще **Q** и Ввод (Enter или Return). (Другой вариант - выполните команду **q()**. в оболочке.) Теперь вы можете прямо перейти к разделу **2.3 Код программ в данной книге, на странице 23.**

Если же, вместо этого, вы получили ошибку о том, что `erl` это неизвестная команда, вам надо установить Эрланг на ваш компьютер. А это означает, что вам придется принять решение - хотите ли вы установить готовый бинарный дистрибутив для вашей машины, воспользоваться пакетным дистрибутивом (на OS X), собрать Эрланг из исходных кодов, или использовать Comprehensive Erlang Archive Network (CEAN) (`\~/Полный сетевой архив Эрланга`)?

Бинарные дистрибутивы

Бинарные дистрибутивы Эрланга доступны для Windows и Linux операционных систем. Инструкции по их установки значительно зависят от конкретной операционной системы. Так что мы пройдемся по им обоим.

Windows

Список релизов вы найдете по адресу <http://www.erlang.org/download.html>. Выбирайте самый последний релиз и кликайте на линке к бинарному дистрибутиву для Windows - он указывает на исполняемый файл. Далее следует стандартная инсталляция для Windows, которая не должна вас вызвать никаких проблем.

Linux

Бинарный пакет существует, например, для Debian варианта Linux. На Debian системе наберите следующую команду:

```
> apt-get install erlang
```

Инсталляция для Mac OS X

Будучи пользователем Mac вы можете инсталлировать готовую версию Эрланга используя систему MacPorts, либо же вы можете собрать Эрланг из исходных кодов. Использовать MacPorts немного проще и она следит за новыми версиями ПО. С другой стороны, MacPorts может несколько запаздывать с релизами Эрланга. Например, во

время написания данной книги версия Эрланга в MacPorts отставала на два релиза от его текущей версии. По этой причине, я рекомендую вам стиснуть зубы и установить Эрланг из его исходного кода, как это описано в следующем разделе. Для этого вам надо убедиться, что у вас установлены средства разработчика (они есть на DVD с ПО, который приходит вместе с вашей машиной).

Сборка Эрланга из исходного кода

Альтернативным способом к инсталляции готовых бинарных дистрибутивов является сборка Эрланга из исходных кодов. Для Windows в этом нет особого смысла, так как каждая версия Эрланга выходит с полными бинарными дистрибутивами для этой ОС, включающими в себя, также, и исходные коды.

Но для пользователей Маков и Linux возможны задержки между официальным релизами Эрланга и готовностью бинарных дистрибутивов для данных систем. Для всех Unix-подобных ОС инструкции по инсталляции одни и те же:

Загрузите последние исходники Эрланга (с адреса <http://www.erlang.org/download.html>). Они будут находиться в файле с названием подобным otp_src_R11B-4.tar.gz (конкретно этот файл содержит четвертый релиз 11-ой версии Эрланга).

Распакуйте, сконфигурируйте, соберите и инсталлируйте согласно следующему:

```
$ tar -xzf otp_src_R11B-4.tar.gz
$ cd otp_src_R11B-4
$ ./configure
$ make
$ sudo make install
```

Примечание: Вы можете использовать команду ./configure - -help чтобы ознакомиться со всеми опциями конфигурации перед построением системы.

Использование CEAN

Полный сетевой архив Эрланга (CEAN) - это попытка собрать вместе основные приложения Эрланга в одном месте с одним инсталлятором. Преимущество в использовании CEAN состоит в том, что там содержатся не только базовые системы Эрланга, но и большое число программных пакетов написанных на Эрланге. Это означает, что не только ваша версия системы Эрланг будет современной, но вы также сможете управлять и своими пакетами.

CEAN содержит готовые бинарные дистрибутивы для большого числа операционных систем и процессорных архитектур. Чтобы инсталлировать систему при помощи CEAN, зайдите на <http://cean.process-one.net/download/> и следуйте инструкциям. (Отметим, что

некоторые пользователи сообщали, что Cean не всегда инсталлирует компилятор Эрланга. Если это произойдет у вас, то запустите оболочку Эрланга и дайте там команду `cean:install(compiler)` - она инсталлирует компилятор.)

2.3. Код программ в данной книге

Большинство примеров кода, которые мы будем приводить в данной книге, вы можете свободно загрузить из сети (по адресу <http://pragmaticprogrammer.com/titles/jaerlang/code.html>). Чтобы помочь вам в этом, некоторые примеры будут снабжены специальной ссылкой вверху, подобно следующему: <http://media.pragprog.com/titles/jaerlang/code/shop1>.

```
-module/shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([]) -> 0.
```

Эта ссылка будет содержать адрес к исходному коду, который можно загрузить себе. Если вы читаете PDF версию данной книги и ваша программа для чтения PDF-файлов поддерживает гиперлинки, то вы можете кликнуть по этой ссылке и указанный код должен появиться в окне вашего браузера.

2.4. Запуск оболочки Эрланга

Теперь давайте начнем. Мы можем взаимодействовать с Эрлангом используя интерактивное средство называемое *Оболочка (Shell)*. Если мы ее запустим, мы можем набирать в ней выражения, а оболочка будет показывать их значения.

Если вы уже инсталлировали Эрланг на вашей машине (раздел 2.2 *Инсталляция Эрланга*), то значит и его оболочка, `erl`, также инсталлирована. Чтобы ее запустить, откройте стандартную командную консоль вашей операционной системы (cmd в Windows или что-то вроде bash в Unix-подобных системах). И запустите оболочку Эрланга, набрав команду `erl`:

```
1. $ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]
Eshell V5.5.1 (abort with ^G)
2. 1> % I'm going to enter some expressions in the shell ..
3. 1> 20 + 30.
```

4. 50
5. 2>

Давайте посмотрим, что мы только что сделали:

1. Эта Unix команда запускает оболочку Эрланга. Оболочка отвечает стандартным приветствием, где говорится какую именно версию Эрланга вы запустили.
2. Оболочка вывела приглашение 1> и мы напечатали комментарий. Знак процента (%) означает начало комментария в языке Эрланг. Любой текст от знака процента до конца строки считается комментарием и игнорируется оболочкой или компилятором Эрланга.
3. Оболочка повторяет приглашение 1> поскольку мы не ввели полной команды. На этот раз мы вводим выражение 20+30 с точкой и возвратом каретки. (Начинающие изучение Эрланга часто забывают ставить точку. Но без нее Эрланг не может определить, что закончили мы наше выражение и хотим увидеть его результат.)
4. Оболочка вычисляет введенное выражение и печатает его результат - 50, в данном случае.
5. Оболочка выводит следующее приглашение, на этот раз, для команды номер 2 (поскольку номер команды увеличивается каждый раз, когда вводится очередная команда).

Вы уже попробовали запустить оболочку Эрланга на вашей машине? Если нет, то, пожалуйста, остановитесь и попробуйте это сделать сейчас. Если вы будете просто читать текст без опробования команд, вы может и будете думать, что все понимаете, но вы не будете переносить ваши знания из вашего мозга в ваши пальцы - программирование это не спорт для зрителей. Как и в любом виде атлетики, вам нужно очень много практиковаться.

Введите выражения из примеров в точности так, как они приведены в тексте книги, а потом немного измените их. Если они не сработают, остановитесь и спросите себя, что пошло не так. Даже опытные программисты на Эрланге проводят множество часов работая с его оболочкой.

По мере накопления вами опыта, вы узнаете, что оболочка - это, на самом деле, очень мощное средство для работы. Предыдущие введенные команды могут быть заменены (с помощью Ctrl+P и Ctrl+N) и отредактированы (командами подобными Emacs-совским). Подробнее об этом рассказано в разделе 6.5 *Редактирование команд в оболочке Эрланга*. А лучше всего вы это поймете, когда начнете писать распределенные программы и узнаете, что можно присоединить вашу оболочку к другой, работающей Эрланг системе, на другом Эрланг узле в кластере или, даже, организовать зашифрованное взаимодействие (ssh) с Эрланг системой работающей на другом, удаленном компьютере. Используя ее вы можете взаимодействовать с любой Эрланг программой на любом Эрланг узле в сообществе таких узлов.

Предупреждение: Вы не можете напечатать в оболочку все, что вы увидите в этой книге. В частности, вы не можете напечатать в оболочку код из примеров Эрланг программ. Синтаксическая форма .erl файла это вовсе не выражение и оболочкой они не воспринимаются. Оболочка может выполнять только Эрланг выражения и не понимает больше ничего другого. В частности нельзя ввести в оболочку заголовок модуля, это те его части, которые начинаются на тире (такие как -module, -export и так далее).

Остальная часть данной главы построена в форме коротких диалогов с оболочкой Эрланга. Часто я не буду рассказывать в мельчайших подробностях, что там происходит, поскольку это помешает изложению материала в книге. В разделе 5.4 *Многочисленные короткие заметки* я дополню некоторые детали.

2.5. Простая арифметика целых чисел

Давайте вычислим несколько простых арифметических выражений:

```
1> 2 + 3 * 4.  
14  
2> (2 + 3) * 4.  
20
```

Важно: Вы видите что диалог начинается с приглашения с номером 1 (то есть оболочка напечатала 1>). Это означает, что мы запустили новую оболочку Эрланга. Всякий раз, когда вы будете видеть, что диалог начинается с 1> вам нужно будет запускать новую оболочку если вы хотите в точности воспроизвести пример из книги. А когда пример начинается с приглашения, номер которого больше чем единица, это означает что работа оболочки продолжена с предыдущего примера и вам не надо запускать ее заново.

Оболочка Эрланг не отвечает?

Если оболочка не отвечает после того, как вы ввели команду, тогда, возможно, вы забыли набрать в конце точку и возврат каретки (что еще называют *точка-конец-строки*).

Также возможно, что вы начали вводить что-то в кавычках (то есть открыли одиночные двойные кавычки (")), но пока не ввели соответствующий им второй символ кавычек, который должен быть точно таким же как и первый.

Если подобное возможно, то лучшее, что вы можете сделать - это ввести закрывающие кавычки и точку-конец-строки.

Если же все серьезно испортилось и система вообще вам не отвечает, тогда нажмите Ctrl+C (в Windows - Ctrl+Break). Вы увидите следующее:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded  
(v)ersion (k)ill (D)b-tables (d)istribution
```

А теперь просто нажмите **A** чтобы прервать текущую сессию работы с Эрлангом.

Подробнее: Вы можете запускать и останавливать множество оболочек. Смотрите подробности в разделе 6.7 *Если оболочка не отвечает.*

Примечание: Если вы хотите опробовать все приводимые примеры из книги в оболочке Эрланга сразу по мере чтения (что является абсолютно наилучшим способом обучения) тогда вам возможно стоит быстро ознакомиться с разделом 6.5 *Редактирование команд в оболочке Эрланга.*

Вы увидите, что Эрланг следует нормальным правилам арифметических выражений, и $2+34$ означает $2+(34)$ а не $(2+3)*4$.

Эрланг использует целые числа произвольного размера для целочисленных вычислений. Целочисленная арифметика в Эрланге точная, так что вам не надо волноваться насчет переполнения или неспособности представить целое большой длины.

Почему бы это не попробовать? Вы можете удивить своих друзей вычисляя очень большие числа:

```
3> 123456789 * 987654321 * 112233445566778899 * 998877665544332211.  
13669560260321809985966198898925761696613427909935341
```

Вы можете вводить целые множеством различных способов (подробнее см. в разделе 5.4 *Целые*) Вот вам пример записи чисел с использованием оснований 16 и 32:

```
4> 16#cafe * 32#sugar.  
1577682511434
```

2.6. Переменные

Нотация для переменных

Часто нам придется говорить о значении какой-то конкретной переменной. Для этого я буду использовать запись вида Var -> Value . То есть, например, запись A -> 42 будет

означать что переменная A имеет значение 42. Когда будет несколько переменных я буду писать {A -> 42, B -> true...} , что означает что A это 42, B это true и так далее.

Как можно сохранить результат выражения, чтобы использовать его в дальнейшем? Для этого предназначены переменные. Вот пример:

```
1> X = 123456789.  
123456789
```

Что здесь произошло? Во-первых, мы присвоили значение переменной X . Потом оболочка Эрланг напечатала ее значение.

Примечание: Все имена переменных должны начинаться с большой буквы.

Если вы хотите узнать значение переменной - просто введите ее имя:

```
2> X.  
123456789
```

Теперь переменная X имеет значение и им можно воспользоваться:

```
3> X*X*X*X.  
232305722798259244150093798251441
```

Тем не менее, если вы попытаетесь присвоить переменной X другое значение, вы получите довольно большое сообщение об ошибке:

```
4> X = 1234.  
=ERROR REPORT===== 11-Sep-2006::20:32:49 ===  
Error in process <0.31.0> with exit value:  
{ {badmatch,1234}, [{erl_eval,expr,3}] }  
exited: { {badmatch,1234}, [{erl_eval,expr,3}] }
```

Что сейчас произошло с Эрлангом? Чтобы объяснить это, мне придется разрушить два ваших внутренних предположения относительно такой простой записи, как X=1234 :

- Во-первых, X это не переменная, по крайней мере не такая переменная, к которым вы привыкли в таких языках как C и Java.
- Во-вторых, = - это вовсе не оператор присвоения.

Возможно это самая сложная область для освоения для всех новичков в Эрланге, так что давайте потратим на нее несколько страниц, чтобы во всем досконально разобраться.

Переменные не изменяются

В Эрланге переменным можно присваивать значения только один раз. Другими словами Эрланг - это язык с *однократно присваиваемыми переменными*. Если вы попытаетесь изменить значение переменной, после того как оно было установлено, вы получите сообщение об ошибке (на самом деле, это сообщение об ошибке несоответствия, как вы могли видеть). Переменная, которой уже назначено значение называется *связанной переменной*. В противном случае она называется *свободной переменной*. Изначально, все переменные свободны.

Когда Эрланг видит выражение типа $X=1234$ он пытается привязать переменной X значение 1234. До такого связывания переменная X может принять любое значение, она - это просто пустая ячейка, которую нужно заполнить. Но, как только, она получает свое значение, оно останется там неизменным навечно.

Одиночное присвоение подобно школьной Алгебре

Когда я ходил в среднюю школу мой учитель математики говорил нам: "Если есть несколько X в разных частях одного уравнения, все эти X означают одно и то же". Именно поэтому мы можем решать уравнения: если мы знаем, что $X+Y=10$ а $X-Y=2$, то X будет 6, а Y будет 4 в обоих этих уравнениях.

Но когда я стал изучать мой первый язык программирования мы увидели вот такую фигню:

```
X=X+1
```

Мы все протестовали, говорили "так нельзя делать!", но учитель программирования сказал, что мы неправы и должны забыть все, что мы изучили на математике. X теперь это вовсе не математическая переменная. Она теперь такая ячейка, подобная маленькой коробочке...

В Эрланге переменные точно такие же, как и в математике. Когда вы присваиваете им значение, вы делаете утверждение - заявляете о непреложном факте. Эта переменная имеет именно это значение. И никак иначе.

Возможно вы теперь удивитесь, а почему мы тогда используем, вообще, слово переменная? На это есть две причины:

Они действительно переменные, но их значение можно изменить только один раз (тогда они переходят из свободного состояния в связанное со значением).

Они выглядят как переменные обычных языков программирования, поэтому когда вы видите строку вида $X=...$ наш мозг говорит нам: "Ага! Я знаю, что это такое. X - это

переменная, а = - это оператор присваивания". И наш мозг почти что прав: X - это почти переменная, а = - это почти что оператор присваивания. (*Примечание: Использование троеточия в коде Эрланга означает просто "код, который я не показываю".*)

На самом деле, = - это оператор сопоставления по образцу, который ведет себя как оператор присвоения, в случае, когда X - это несвязанная (свободная) переменная.

И, наконец, *область видимости переменной* - это всегда лексический раздел кода, в котором она была определена. Так что, если X использовалась внутри определения варианта функции, ее значение никак "не убежит" из этих границ. В Эрланге не существует таких вещей, как глобальные или приватные переменные используемые многими вариантами одной функции. Если X появляется в разных функциях, это все разные X.

Соответствие Образцу

В большинстве языков символ = означает оператор присваивания. Тем не менее в Эрланге символ = означает операцию *сопоставления по образцу*. Выражение L = P означает на самом деле следующее: вычисляется правая сторона (P) и, потом, она сопоставляется образцу в левой стороне выражения (L).

Переменная, такая, например, как X, является простейшей формой образца (или паттерна). Как мы уже говорили ранее, переменной можно присваивать ее значение только один раз. Когда мы *первый* раз говорим X=ПервоеВыражение, Эрланг спрашивает сам себя: "Что я могу такого сделать, чтобы это утверждение было истинным?" Поскольку X пока не имеет никакого значения, то можно привязать X к значению Первого Выражения, и тогда это утверждение все станет корректным, и все будут счастливы.

Если потом, когда нибудь, мы скажем X=ДругоеВыражение, то это будет корректным только если ПервоеВыражение и ДругоеВыражение идентичны. Вот вам пример всего этого:

```
1> X = (2+4).
6
2> Y = 10.
10
3> X = 6.
6
4> X = Y.
=ERROR REPORT===== 27-Oct-2006::17:25:25 ===
Error in process <0.32.0> with exit value:
  [{badmatch,10},[{erl_eval,expr,3}]}]
```

```
5> Y = 10.  
10  
6> Y = 4.  
=ERROR REPORT==== 27-Oct-2006::17:25:46 ===  
Error in process <0.37.0> with exit value:  
{{badmatch,4},[{erl_eval,expr,3}]}  
7> Y = X.  
=ERROR REPORT==== 27-Oct-2006::17:25:57 ===  
Error in process <0.40.0> with exit value:  
{{badmatch,6},[{erl_eval,expr,3}]}
```

Вот что здесь произошло: В строке 1 система вычислила значение $2+4$ и ответила что это будет 6. После этой строки оболочка имеет следующее множество связанных переменных: $\{X \rightarrow 6\}$. А после третьей строки мы будем иметь следующее множество связей: $\{X \rightarrow 6, Y \rightarrow 10\}$.

Теперь мы добрались до строки 5. Еще до вычисления выражения мы знали что $X \rightarrow 6$ и, значит, сопоставление по образцу $X = 6$ проходит вполне успешно.

Потом мы говорим, что $X = Y$ в строке 7. Но множество наших связей это $\{X \rightarrow 6, Y \rightarrow 10\}$, а значит, это сопоставление образцов не проходит и нам выдается сообщение об ошибке.

Утверждения в строках с 4 по 7-ую либо заканчиваются успешно, либо нет, в зависимости от значений X и Y . Теперь самое время изучить их всех хорошенько и убедиться, что вы все их отлично понимаете, перед тем как читать что-то далее.

На этой стадии вам может показаться, что я вам втолковываю очевидное. Все паттерны слева от " $=$ " - это просто переменные, либо связанные, либо нет. Но как мы увидим далее мы можем там поставить очень сложные паттерны и сопоставить их с помощью оператора " $=$ ". Мы с вами вернемся к этой теме после того как познакомимся с кортежами и списками, которые используются для хранения сложных, составных структур данных.

Почему одиночное присвоение делает мои программы лучше?

В Эрланге переменная - это просто ссылка на ее значение. В имплементации Эрланга связанная переменная представлена указателем на область хранения, содержащую значение. И это значение нельзя изменить.

То, что никак нельзя изменить значение связанной переменной, это очень важно и очень необычно, по сравнению с императивными языками программирования, такими как Си и Java.

Давайте посмотрим, что бы могло произойти, если бы было разрешено менять

переменные. Давайте определим переменную X следующим образом:

```
1> X = 23.  
23
```

Теперь используем X в вычислениях:

```
2> Y = 4 * X + 3.  
95
```

А теперь, допустим, что нам можно изменить значение X:

```
3> X = 19.
```

К счастью, Эрланг этого не позволяет. Его компилятор начинает орать как ненормальный и выдает что-то вроде вот этого:

```
=ERROR REPORT==== 27-Oct-2006::13:36:24 ===  
Error in process <0.31.0> with exit value:  
{[badmatch,19],[{erl_eval,expr,3}]}
```

Что просто означает, что X не может быть 19, раз уж мы определили, что она будет 23.

Но, давайте просто допустим, что мы можем сделать это. Тогда значение Y становится тоже некорректным, в том смысле, что мы уже не можем полагать утверждение за номером 2 как уравнение. Более того, если X может менять свое значение много раз в программе и что-то в ней пошло не так, может потребоваться много усилий, чтобы понять, какое именно значение X привело к ошибке и в какой именно точке программы вдруг появилось это неверное значение.

В Эрланге значения переменных не могут быть изменены, после того как они были установлены. Это упрощает поиск ошибок (debugging). Чтобы понять, почему это так, мы должны спросить сами себя, а что такое ошибка, вообще, и как мы узнаем о ней.

Одним из наиболее распространенных способов определить, что программа работает не корректно, является обнаружение, что ее переменная приняла неверное значение. В этом случае, вам придется искать в программе то место, где эта переменная получает свое неправильное значение. Если эта переменная меняется много-много раз и в различных местах вашей программы, тогда такие поиски места некорректного изменения ее значения, могут оказаться весьма и весьма трудными.

В Эрланге таких проблем просто нет. Переменная может быть установлена только один раз и более никогда не меняется. Так что если мы обнаруживаем, что у

переменной неправильное значение, мы немедленно переходим к месту где эта переменная стала связанный с этим значением, а значит именно там и произошла ошибка.

Возможно вы сейчас задаетесь вопросом, а как, вообще, возможно писать программы без переменных? Как можно выразить что-то подобное $X=X+1$ средствами Эрланга? Ответ очень прост. Придумайте новую переменную, чье имя еще не было использовано (например, $X1$) и напишите $X1=X+1$.

Отсутствие побочных эффектов означает возможность распараллелить нашу программу

Технически, области памяти, которые могут быть модифицированы, называются *изменяемым состоянием*. Эрланг - это язык программирования с неизменяемыми состояниями.

Гораздо позднее в этой книге мы будем рассматривать вопросы программирования многоядерных процессоров. Когда дело доходит до этого, последствия отсутствия изменяемых состояний, просто огромные.

Если вы используете обычный язык программирования, такой как Си или Java для программирования многоядерных процессоров, тогда вы столкнетесь в проблемой, называемой *разделяемой памятью*. Чтобы не испортить эту разделяемую память, она должна быть заблокирована во время ее использования. Поэтому программы, пользующиеся ей, не должны зависать или падать, во время ее использования.

В Эрланге нет изменяемых состояний, нет разделяемой памяти и нет блокировок. Это позволяет гораздо легче распараллеливать наши программы.

2.7. Числа с плавающей точкой

Давайте попробуем несколько арифметических примеров с вещественными числами:

```
1> 5/3.  
1.66667  
2> 4/2.  
2.00000  
3> 5 div 3.  
1  
4> 5 rem 3.  
2  
5> 4 div 2.  
2
```

```
6> Pi = 3.14159.  
3.14159  
7> R = 5.  
5  
8> Pi * R * R.  
78.5397
```

Обратите внимание. В строке 1 число в конце выражения - это целое число 3. Точка означает конец выражения, а не десятичную точку. Если бы я хотел ввести здесь число с плавающей точкой, то надо было ввести 3.0.

"/" всегда возвращает вещественное число. Поэтому $4/2$ равно 2.0000 (в оболочке Эрланга). N div M и N rem M используются для целочисленного деления и остатка. Поэтому $5 \text{ div } 3$ дает 1, а $5 \text{ rem } 3$ дает 2.

Вещественные числа должны иметь десятичную точку и, по крайней мере, одну цифру после нее. Когда вы делите два целых числа с помощью "/", то результат автоматически конвертируется в число с плавающей точкой.

2.8. Атомы

В Эрланге атомы используются для представления различных не-числовых констант.

Если вы использовали перечислимые типы в Си или Java, то вы уже использовали что-то очень похожее на атомы, сами того не подозревая.

Программисты на Си должны быть знакомы с соглашением об использовании символьических имен констант, чтобы программа была более само-документирована. В типичной Си-программе определяется множество глобальных констант во включаемом файле, который состоит из большого числа их определений. Например, некоторый файл glob.h может содержать в себе:

```
#define OP_READ 1  
#define OP_WRITE 2  
#define OP_SEEK 3  
...  
#define RET_SUCCESS 223  
...
```

Типичный Си код, использующий эти символьические константы, может выглядеть следующим образом:

```
#include "glob.h"
```

```
int ret;
ret = file_operation(OP_READ, buff);
if( ret == RET_SUCCESS ) { ... }
```

В Си программах сами значения этих констант часто совсем не интересны. От них важно только, чтобы они были все разные и их можно было сравнивать на равенство и неравенство.

В Эрланге эквивалент этой программы может выглядеть следующим образом:

```
Ret = file_operation(op_read, Buff),
if Ret == ret_success ->
...
```

В Эрланге все атомы глобальны и это достигается без определений макросов или включаемых файлов.

Предположим, вы хотите написать программу, которая манипулирует днями недели. Как вы представите дни недели в Эрланге. Конечно же вы используете атомы `monday`, `tuesday`, ...

Все атомы начинаются с прописной буквы, а далее может быть буквенно-числовая последовательность, включающая в себя подчеркивание (`_`) или знак `@`. Например: `red`, `december`, `cat`, `meters`, `yards`, `joe@somehost`, and `a_long_name`. (Вы можете обнаружить, что точка `(.)` тоже может использоваться в атомах - это одна из неофициальных возможностей Эрланга.)

Атомы также могут быть заключены в одиночные кавычки `('')`. Используя такую их форму мы можем создавать атомы, которые начинаются с большой буквы (которые, иначе, считались бы переменными) или содержащие не только буквенно-числовые символы. Например: `'Monday'`, `'Tuesday'`, `'+'`, `'*'`, `'an atom with spaces'`. Вы также можете поместить в одиночные кавычки и обычный атом, при этом `'a'` будет в точности тем же самым что и просто `a`.

Значением атома является сам атом. Так что если вы дадите команду, которая состоит только из одного атома, то Эрланг и выведет значение этого атома.

```
1> hello.
hello
```

Может показаться странным, что мы обсуждаем значения атомов или значения целых чисел. Но, надо не забывать, что Эрланг - это функциональный язык программирования, в котором любое выражение должно иметь свое значение. Это

включает в себя и целые числа и атомы, которые являются просто крайне примитивными формами выражений.

2.9. Кортежи

Предположим вам нужна группа из фиксированного числа объектов как единая сущность. Для этого мы будем использовать кортеж (tuple). Вы можете легко создать кортеж просто заключив нужные вам значения в фигурные скобки и разделив их запятыми. Например, если вы хотите записать чье-либо имя и рост, вы можете использовать { joe, 1.82 }. Этот кортеж содержит атом и число с плавающей точкой.

Кортежи похожи на структуры в Си, но при условии, что они анонимные. В Си переменная P типа точка может быть определена следующим образом:

```
struct point {  
    int x;  
    int y;  
} P;
```

Доступ к полям этой структуры осуществляется в Си с использованием оператора точка. Так что для установки значений x и у в у этой переменной вы можете написать:

```
P.x = 10; P.y = 45;
```

В Эрланге нет определения типов, чтобы создать "точку" и мы можем написать просто:

```
P = {10, 45}
```

Эта запись создает кортеж и привязывает его к переменной P. В отличии от Си поля кортежа не имеют имен. И поскольку кортеж содержит в себе только несколько целых чисел, мы должны сами помнить для чего они используются. Чтобы облегчить такое запоминание, обычной практикой является использование атома в качестве первого элемента кортежа, который описывает, что представляет собой этот кортеж. То есть мы лучше напишем {point, 10, 45}, а не {10, 45}, что сделает нашу программу намного более удобочитаемой. (Это, конечно, вовсе не требование языка, а просто рекомендуемый стиль программирования.)

Кортежи могут быть вложенными друг в друга. Предположим мы хотим представить некоторые факты о человеке - его имя, рост, размер ноги и цвет глаз. мы можем сделать это следующим способом:

```
1> Person = {person,
    {name, joe},
    {height, 1.82},
    {footsize, 42},
    {eyecolour, brown}}.
```

Заметьте, что для цвета глаз (в последней строчке) мы использовали атомы как для обозначения имени поля кортежа, так и для задания его значения.

Создание кортежей

Кортежи создаются автоматически, когда их объявляют в программе и автоматически уничтожаются, если их больше не используют. В Эрланге используется сборщик мусора для освобождения неиспользуемой памяти, так что нам не надо беспокоиться насчет выделения и освобождения памяти.

Если при построении кортежа вы используете переменную, тогда он будет разделять с ней то же значение данных, на структуру которых она и указывает. Приведем пример:

```
2> F = {firstName, joe}.
{firstName,joe}
3> L = {lastName, armstrong}.
{lastName,armstrong}
4> P = {person, F, L}.
{person,{firstName,joe},{lastName,armstrong}}
```

Если вы попробуете создать структуру данных с неопределенной (т.е. несвязанной) переменной, вы получите ошибку. Рассмотрим это также на примере, используя неопределенную переменную Q:

```
5> {true, Q, 23, Costs}.
1: variable 'Q' is unbound
```

Это просто означает, что переменная Q не определена.

Извлечение данных из кортежа

Ранее мы говорили, что "=" , которое выглядит как оператор присваивания, на самом деле таковым не является, а представляет собой, на самом деле, оператор сопоставления по образцу. Вы тогда еще могли удивиться, и зачем это нам надо было быть такими педантичными. Ну, дело в том, что сопоставление по образцу — это фундаментальный механизм в Эрланге и он используется в нем для множества разных вещей. Он используется для извлечения данных из различных структур данных, а также для контроля порядка исполнения внутри функции и для выбора, какое из

пришедших сообщений обработать в параллельных программах из числа посланных данному процессу.

Если нам нужно извлечь какие-то данные из кортежа, то мы также будем использовать оператор сопоставления "=". .

Давайте вернемся к нашему кортежу, который представляет собой точку на экране:

```
1> Point = {point, 10, 45}.
```

```
{point, 10, 45}.
```

Предположим мы хотим из него извлечь данные из его полей в две переменные X и Y. Поступим следующим образом:

```
2> {point, X, Y} = Point.
```

```
{point,10,45}
```

```
3> X.
```

```
10
```

```
4> Y.
```

```
45
```

В команде под номером 2 переменная X привязывается к 10, а Y к 45. Значением выражения L=П является значение П (правой части) , поэтому оболочка печатает {point,10,45}.

Как можно видеть, кортежи с обоих сторон знака равенства должны иметь одинаковое число элементов и соответствующие элементы с обоих сторон должны иметь одинаковое значение.

А теперь предположим, что вы ввели что-то вроде вот этого:

```
5> {point, C, C} = Point.
```

```
=ERROR REPORT===== 28-Oct-2006::17:17:00 ===
```

```
Error in process <0.32.0> with exit value:
```

```
{badmatch,{point,10,45}},[{erl_eval,expr,3}]}
```

Что же тут произошло? Образец {point, C, C} не может соответствовать {point, 10, 45} так как C не может быть одновременно и 10 и 45. Поэтому сопоставление по образцу не удалось и система выдала ошибку. (Для читателей знакомых с Прологом: Эрланг считает несоответствие образцов ошибкой и не имеет отката (backtrack).)

Если у вас имеется сложный кортеж и вы хотите извлечь из него данные, то вы можете написать кортеж такой же формы (структуры) как и ваш, но в тех местах откуда вы хотите извлечь данные поместите несвязанные переменные. (Этот метод

извлечения данных путем сопоставления по образцу называется *унификацией* и используется во множестве функциональных и логических языков программирования.)

Чтобы продемонстрировать все это мы, для начала, определим переменную Person которая будет содержать в себе сложную структуру данных:

```
1> Person={person,{name,{first,joe},{last,armstrong}},{footsize,42}}.  
{person,{name,{first,joe},{last,armstrong}},{footsize,42}}
```

Теперь мы напишем паттерн (образец) для извлечения имени этой персоны:

```
2> {_,{_,{_,Who},_},_} = Person.  
{person,{name,{first,joe},{last,armstrong}},{footsize,42}}
```

И, наконец, мы распечатаем значение Who:

```
3> Who.  
joe
```

Заметьте, что в предыдущем примере мы написали символ _ вместо переменных, которые нам сейчас не интересны. Символ _ называется *анонимной переменной*. В отличии от обычных переменных, несколько _ в одном паттерне, вовсе не привязываются к одному и тому-же значению.

2.10 Списки

Мы используем в жизни списки для хранения самых различных вещей: того, что надо купить в магазине, имен планет, результатов возвращенных вашей функцией простых чисел и так далее.

Мы создаем список в Эрланге путем заключения списка его элементов в квадратные скобки и разделяя их запятыми. Вот как мы можем создать список покупок:

```
1> ThingsToBuy = [{apples,10},{pears,6},{milk,3}].  
[{apples,10},{pears,6},{milk,3}]
```

Сами элементы списка могут быть произвольного типа, так что мы можем, например, написать:

```
2> [1+7,hello,2-2,{cost, apple, 30-20},3].  
[8,hello,0,{cost,apple,10},3]
```

Терминология

Мы называем первый элемент списка *головой* списка. А если удалить из списка его голову, то все оставшееся называется *хвостом* списка.

Например, если у нас есть список [1,2,3,4,5], то его головой будет целое число 1, а хвостом список [2,3,4,5]. Заметьте, что головой списка может быть все что угодно, но хвост списка это, обычно, тоже список.

Доступ к голове списка - это очень быстрая операция, так что чуть ли не все функции обработки списков начинают с выделения его головы, ее обработки, а потом переходят, аналогично к хвосту списка.

Определение списков

Если T - это список то $[H|T]$ это тоже список в котором голова - это H , а хвост это T . Вертикальная черта отделяет голову списка от его хвоста. $[]$ - это пустой список.

Когда бы мы не создавали список с использованием конструктора $[...|T]$, мы должны быть уверены, что T - это список. Если это так, то новый список будет "правильно сформирован". А если T это не список, тогда такой новый список называют "неправильный список". Большинство функций библиотечных функций полагают, что список для них был правильно сформирован и не будут работать с неправильными списками.

Мы можем добавить более одного элемента в начало T если напишем $[E_1, E_2, \dots, E_n | T]$.
Например:

```
3> ThingsToBuy1 = [{oranges,4},{newspaper,1} | ThingsToBuy].  
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Выделение элементов из списка

Как и из всего остального, мы можем извлекать элементы из списка с помощью оператора сопоставления по образцу. Если у нас имеется не пустой список L , тогда выражение $[X|Y] = L$, где X и Y - это несвязанные переменные, поместит голову списка в X , а хвост списка - в Y .

Итак, мы пришли в магазин и у нас собой наш список необходимых покупок - $ThingsToBuy1$. Первым делом нам надо распаковать этот список на голову и хвост:

```
4> [Buy1 | ThingsToBuy2] = ThingsToBuy1.  
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Это успешно связывает

```
Buy1 -> {oranges,4}
```

и

```
ThingsToBuy2 -> [{newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
```

Мы идем и покупаем апельсины, а потом мы хотим извлечь еще несколько позиций:

```
5> [Buy2,Buy3|ThingsToBuy3] = ThingsToBuy2.  
{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Это приводит к тому, что Buy2 -> {newspaper,1}, Buy3 -> {apples,10}, и ThingsToBuy3 -> [{pears,6},{milk,3}].

2.11. Строки

Строго говоря, в Эрланге нет строчек. *Строки*, на самом деле, это просто списки целых чисел. строки заключаются в двойные кавычки (""), так что мы можем, например, написать:

```
1> Name = "Hello".  
"Hello"
```

Примечание: В некоторых языках строки можно заключать и в двойные и в одиночные кавычки. В Эрланге вы можете использовать только двойные кавычки.

"Hello" - это просто краткая запись списка целых чисел, которые представляют отдельные буквы в этой строчке.

Когда оболочка распечатывает значение списка, она печатает список как строку, но только тогда, когда все целые в нем представляют собой печатные буквы:

```
2> [1,2,3].  
[1,2,3]  
3> [83,117,114,112,114,105,115,101].  
"Surprise"  
4> [1,83,117,114,112,114,105,115,101].  
[1,83,117,114,112,114,105,115,101].
```

В выражении номер 2 список [1,2,3] напечатан без изменения. Это потому, что 1, 2 и 3 - это не печатные символы.

В выражении номер 3 все позиции в списке - это печатные символы, поэтому он печатается как строка.

Выражение 4 почти точно такое как и выражение 3, за исключением того что список начинается с 1, что не является печатным символом. Из-за этого весь список печатается без конверсии в строку.

нам совсем не нужно знать, какое целое число представляет конкретный символ. Для этого мы можем использовать "долларовый синтаксис". Например, \$a - это на самом деле целое число, которое представляет собой символ "a" и так далее.

```
5> I = $s.  
115  
6> [I-32,$u,$r,$p,$r,$i,$s,$e].  
"Surprise"
```

Набор символов используемый в строках

Буквы в строке представляются кодами из таблицы Latin-1 (ISO-8859-1). Например, строка содержащая шведское имя Håkan будет закодирована как [72,229,107,97,110].

Примечание: Если вы ведете [72,229,107,97,110] как выражение в оболочке Эрланга, то, возможно, вы не получите то, что вы ожидали:

```
1> [72,229,107,97,110].  
"H\\345kan"
```

Что случилось с "Håkan", куда он подевался? На самом деле это никак не связано с Эрлангом, а только с локальной кодировочной таблицей для вашего терминала.

Эрланг просто считает строку списком целых чисел в какой-то кодировке и более его ничего не волнует. Если они вдруг печатные согласно кодам из Latin-1, тогда они должны корректно отображаться (если корректны установки на вашем терминале).

2.12. Снова о сопоставлении по образцу

В завершении этой главы мы еще раз вернемся к сопоставлению по образцу.

В следующей таблице приведено несколько примеров паттернов (образцов) и терминов (структур данных Эрланга). Третья колонка этой таблицы, называемая

Результат, показывает соответствует ли образец термину и, если это так, то и создаваемые, при этом, связи переменных. Посмотрите на все эти примеры и убедитесь, что вы все их действительно понимаете:

Паттерн	Термин	Результат
{X,abc}	{123,abc}	Успех. X -> 123
{X,Y,Z}	{222,def,"cat"}	Успех. X -> 222, Y -> def, Z -> "cat"
{X,Y}	{333,ghi,"cat"}	Провал. У кортежей разная структура
X	true	Успех. X -> true
{X,Y,X}	{{abc,12},42, {abc,12}}	Успех. X -> {abc,12}, Y -> 42
{X,Y,X}	{{abc,12},42,true}	Провал. X не может быть одновременно и {abc,12} и true
[H T]	[1,2,3,4,5]	Успех. H -> 1, T -> [2,3,4,5]
[H T]	"cat"	Успех. H -> 99, T -> "at"
[A,B,C T]	[a,b,c,d,e,f]	Успех. A -> a, B -> b, C -> c, T -> [d,e,f]

Если вы не уверены в каком то из этих примеров, тогда попытайтесь ввести соответствующее выражение вида Паттерн = Термин в оболочку Эрланга и посмотреть что из этого выйдет.

Например:

```

1> {X, abc} = {123, abc}.
{123,abc}.
2> X.
123
3> f().
ok
4> {X,Y,Z} = {222,def, "cat"}.
{222,def,"cat"}.
5> X.
222
6> Y.
def
...

```

Примечание: команда f(). говорит оболочке Эрланга забыть все связывания, которые у нее были до этого. После этой команды все переменные становятся свободными. Так что X в строке 4 никак не связана с X в строках 1 и 2.

Теперь, когда мы хорошо освоились с базовыми типами данных и с идеями одиночного присвоения и сопоставления по образцу, мы можем немного ускорить темп и рассмотреть как определять функции и модули. Давайте рассмотрим это в следующей главе.

Глава 3. Последовательное программирование

В этой главе мы рассмотрим, как на Эрланге пишутся простые последовательные программы.

В первом разделе мы поговорим о модулях и функциях. Мы увидим как принцип сопоставления шаблонов, о котором мы узнали в предыдущей главе, используется при объявлении функций.

Сразу после этого мы вернемся к списку покупок, который мы составили в предыдущей главе, и напишем некоторый код, который будет подсчитывать общую стоимость этого списка.

По мере продвижения вперед, мы будем постепенно улучшать написанные нами программы. И вы, при этом, будете видеть как программы на Эрланге постепенно развиваются, а не просто получите законченные программы, без объяснения того, как мы их получили. Понимание основных шагов этого процесса, может подсказать вам, как вы можете его применять для своих программ.

Затем мы поговорим о функциях высшего порядка (называемых funs), и о том, как они могут быть использованы для создания ваших собственных управляемых абстракций.

В завершении мы поговорим о часовых (guard), записях, и выражениях `case` и `if`.

Итак, давайте приступим...

3.1. Модули

В Эрланге, модули – это основная единица кода. Все функции, которые мы пишем, содержатся в модулях. Модули Эрланга сохраняются в файлах с расширением `.erl`. Модули должны быть откомпилированы перед тем как их содержимое будет готово к выполнению. Скомпилированный модуль будет иметь расширение `.beam`. (`beam` - это сокращение от "Bogdan's Erlang Abstract Machine" ("Абстрактная машина Эрланга Богдана"). Богумил (Богдан) Хаусман написал компилятор Эрланга в 1993 году и разработал новый набор инструкций для Эрланга)

Перед тем как написать свой первый модуль, давайте вспомним о сопоставлении шаблонов (или образцов). Для начала создадим пару структур данных, представляющих собой прямоугольник и круг. Затем извлечем из этих структур значения длин сторон для прямоугольника и радиуса для круга. Например, вот так:

```
1> Rectangle = {rectangle, 10, 5}.
```

```
{rectangle, 10, 5}.
```

```
2> Circle = {circle, 2.4}.
```

```
{circle,2.40000}
```

```
3> {rectangle, Width, Ht} = Rectangle.
```

```
{rectangle,10,5}
```

```
4> Width.
```

```
10
```

```
5> Ht.
```

```
5
```

```
6> {circle, R} = Circle.
```

```
{circle,2.40000}
```

```
7> R.
```

```
2.40000
```

В строках 1 и 2 мы создали прямоугольник и круг. В строках 3 и 6 мы извлекли "распаковали") значения полей прямоугольника и круга, используя сопоставление шаблонов. В строках 4, 5 и 7 мы выводим значения переменных, которые были получены путем сопоставления шаблонов. После исполнения строки 7, мы имеем следующие связанные переменные: `{Width -> 10, Ht -> 5, R -> 2.4}`.

Перейти от сопоставления шаблонов (образцов) в командной строке к сопоставлению шаблонов в функциях - это совсем несложный шаг. Давайте начнем с написания функции `area` вычисляющей площадь прямоугольника и круга. Мы поместим ее в модуль `geometry`, а модуль сохраним в файле `geometry.erl`. Данный модуль выглядит так:

[Скачать `geometry.erl`](#)

```
-module(geometry).
```

```
-export([area/1]).
```

```
area({rectangle, Width, Ht}) -> Width * Ht;
```

```
area({circle, R}) -> 3.14159 * R * R.
```

Не обращайте пока внимания на директивы `-module` и `-export` (мы поговорим о них позже). сейчас я прошу вас пристально посмотреть на код функции `area`.

Функция `area` содержит два варианта сопоставления аргументов, которые мы будем называть **клаузами** (от англ. слова `clause` = условие, клауза, клаузула (условия

применения в юридическом документе)), либо же "вариантом применения функции". Клаузы между собой разделяются точкой с запятой, а последняя клауза функции завершается точкой.

У каждого варианта применения функции (клаузы) есть свои заголовок и тело; заголовок содержит имя функции и шаблон аргументов (в круглых скобках), а тело состоит из последовательности выражений (См. раздел 5.4 *Выражения и Последовательности выражений*), которые вычисляются если шаблон аргументов в заголовке совпал с реально выданными этой функции аргументами. Сопоставление шаблонов аргументов с реальными аргументами происходит в том-же порядке, в котором были объявлены клаузы функции. Обратите внимание, что шаблон аргументов `{rectangle, Width, Ht}` является частью объявления клаузы функции. Каждый шаблон аргументов определяет только один вариант применения функции (клаузу). Давайте посмотрим на первую клаузу:

```
area({rectangle, Width, Ht}) -> Width * Ht;
```

Это функция вычисления площади прямоугольника. Когда мы вызываем `geometry:area({rectangle, 10, 5})`, сопоставлением шаблонов аргументам присваиваются выданные значения `{Width -> 10, Ht -> 5}`. А после стрелки `->` следует код, который будет потом выполнен. Это просто `Width * Ht`, или же 10×5 , или же 50.

Сейчас мы это скомпилируем и запустим:

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({circle, 1.4}).
6.15752
```

Что здесь произошло? В строке 1 мы выполнили команду `c(geometry)`, которая скомпилировала код в файле (модуле) `geometry.erl`. Компилятор возвратил `{ok,geometry}`, что значит, что компиляция прошла успешно и модуль `geometry` был скомпилирован и загружен. В строках 2 и 3 мы вызываем функции содержащиеся в модуле `geometry`. Обратите внимание, мы должны указывать имя функции вместе с именем модуля, для однозначного указания, какую именно функцию мы хотим вызвать (в разных модулях могут быть функции с одинаковыми именами).

Развитие возможностей программы.

Теперь, допустим, мы хотим дополнить нашу программу, добавив квадрат к нашим

геометрическим объектам. Мы можем написать это так:

```
area({rectangle, Width, Ht}) -> Width * Ht;  
area({circle, R}) -> 3.14159 * R * R;  
area({square, X}) -> X * X.
```

или так:

```
area({rectangle, Width, Ht}) -> Width * Ht;  
area({square, X}) -> X * X;  
area({circle, R}) -> 3.14159 * R * R.
```

В данном случае порядок кауз данной функции не столь важен; для программы не имеет значения в каком порядке они следуют. Потому что шаблоны аргументов в этих клаузах взаимоисключающие. Это делает написание и расширение программ очень простым делом - мы просто добавляем новые варианты применения со своими шаблонами аргументов к нашей функции. Однако, в общем случае, порядок следования клауз с шаблонами аргументов имеет значение. При вызове функции, аргументы, с которыми была вызванна функция, сопоставляются с шаблонами ее применения в том порядке, в котором они были объявлены в файле.

Перед тем, как мы пойдем дальше, вы должны обратить внимание на некоторые детали реализации функции area:

- Функция area содержит несколько клауз. Когда мы вызываем функцию, выполнение начинается с первой клаузы, совпавшей с аргументами, с которыми функция была вызвана.
- Наша функция не предусматривает случай, когда не одна из клауз не совпадет с аргументами - тогда наша программа завершится с ошибкой времени исполнения. Это сделано нами преднамеренно.

Куда подевался мой код?

Если вы загрузили примеры кода из этой книги или хотите написать свои собственные примеры, вам нужно убедиться, что, когда вы запускаете компиляцию из оболочки Эрланга, вы находитесь в директории, в которой система сможет найти ваш код.

Если вы запустили оболочку Эрланга из командной строки, то вам будет необходимо изменить текущую директорию на ту, в которой содержится ваш код, перед тем, как пытаться его скомпилировать.

Если вы запускаете оболочку Эрланга в Windows, использую стандартный Erlang

дистрибутив, вам также следует изменить директории на те, в которых вы храните ваш код. Две команды в Эрланге помогут вам перейти в нужную директорию. Если вы заблудились, то команда `pwd()` напечатает вам текущую директорию оболочки Эрланга. А команда `cd(Dir)` изменит текущую рабочую директорию на директорию `Dir`. Однако Вы должны использовать прямые слеши в имени директории (даже в Windows); например:

```
1> cd("c:/work" ).  
c:/work
```

Подсказка пользователям Windows: Создайте файл `C:/Program Files/erl5.4.12/.erlang` (измените его путь, если в вашей системе он отличается). И добавьте следующие команды в этот файл:

```
io:format("consulting .erlang in \~p\~n",  
[element(2,file:get_cwd())]).  
% Edit to the directory where you store your code  
c:cd("c:/work" ).  
io:format("Now in:\~p\~n" , [element(2,file:get_cwd())]).
```

Теперь, когда вы запустите оболочку Эрланга, текущая директория автоматически изменится на `C:/work`, или ту, которую вы укажите.

(Прим. переводчика: с русскими буквами в Эрланге могут быть проблемы. Везде.)

Многие языки программирования, такие как Си, содержат только одну точку входа в свои функции. Поэтому, если мы захотим реализовать нашу функцию на Си, то аналогичный код будет выглядеть примерно следующим образом:

```
enum ShapeType { Rectangle, Circle, Square };  
  
struct Shape {  
    enum ShapeType kind;  
    union {  
        struct { int width, height; } rectangleData;  
        struct { int radius; } circleData;  
        struct { int side; } squareData;  
    } shapeData;  
};  
  
double area(struct Shape* s) {  
    if( s->kind == Rectangle ) {  
        int width, ht;  
        width = s->shapeData.rectangleData.width;
```

```
    ht = s->shapeData.rectangleData.ht;
    return width * ht;
} else if ( s->kind == Circle ) {
...
}
```

Данный Си-код, по-сути, также выполняет операцию сопоставления шаблонов реальным аргументам функции. Но только в данном случае, программист должен сам написать код реализующий все это, да еще и убедиться, что он работает корректно.

В Эрланге же мы просто пишем шаблоны аргументов в клаузах функций, а компилятор генерирует оптимальный код для сопоставления шаблонов, который всегда выбирает корректный вариант.

Мы можем также рассмотреть, как будет выглядеть аналогичный код написанный на Java (в духе объектно-ориентированного программирования):

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double ht;
    final double width;

    Rectangle(double width, double height) {
        this.ht = height;
        this.width = width;
    }

    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;

    Square(double side) {
        this.side = side;
    }
}
```

```
    double area() { return side * side; }
}
```

Если сравнить этот код с кодом написанный на Эрланге, то можно легко заметить, что в Java-программе код для вычисления площади находится в трех различных местах, а в Erlang-программе этот код находится компактно, в одном месте.

3.2. Вернемся к шоппингу

Напомню, что у нас имеется список покупок, который выглядит вот так (*{предмет,количество}*):

```
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Теперь предположим, что мы хотим узнать сколько будут стоить все наши продукты. Для этого мы должны знать какова стоимость каждого элемента из списка покупок. Давайте соберем эту необходимую информацию в модуле с именем `shop`. Запустите свой любимый текстовый редактор и введите код, приведенный ниже, в файл с именем `shop.erl`.

[Скачать shop.erl](#)

```
-module(shop).
-export([cost/1]).
cost(oranges) -> 5;
cost(newspaper) -> 8;
cost(apples) -> 2;
cost(pears) -> 9;
cost(milk) -> 7.
```

Функция `cost/1` (что означает: функция `cost` с одним аргументом) состоит из 5 клауз. Заголовок каждой клаузы содержит шаблон аргументов (в данном случае это простейший шаблон, состоящий из одного атома). Когда мы вызовем `shop:cost(X)` система попытается сопоставить `X` с каждым шаблоном в этих клаузах функции `cost`. Если совпадение будет найдено, выполнится код следующий в этой клаузе вслед за символом `->`.

Кроме того, функция `cost/1` должна быть экспортирована из своего модуля; это необходимо если мы хотим вызывать ее вне модуля. (Также можно написать `-compile(export_all).`, что экспортирует все функции, содержащиеся в модуле с такой директивой).

Давайте протестируем все это. Мы скомпилируем и запустим нашу программу из оболочки Эрланга:

```
1> c(shop).
{ok,shop}
2> shop:cost(apples).
2
3> shop:cost(oranges).
5
4> shop:cost(socks).
=ERROR REPORT==== 30-Oct-2006::20:45:10 ===
Error in process <0.34.0> with exit value:
  {function_clause,[{shop,cost,[socks]}, {erl_eval,do_apply,5}, {shell,exprs,6}, {shell,eval_loop,3}]}
```

В строке 1 мы компилируем модуль из файла с именем `shop.erl`. В строках 2 и 3 мы проверяем сколько стоят яблоки и апельсины (результат: 2 и 5 денежных единиц (не важно каких)). В строке 4 мы запрашиваем стоимость носок, но подобного варианта применения (клаузы) в `cost` нет, и поэтому, мы получаем ошибку сопоставления шаблонов, о чём система выводит нам сообщение. (Пометка `function_clause` в сообщении об ошибке, как раз и говорит нам о том, что вызов функции оказался невозможным, поскольку ни одна из ее клауз не подошла переданным аргументам.)

Вернемся к нашему списку. Предположим мы имеем следующий список покупок:

```
1> Buy = [{oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
[ {oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3} ]
```

И, допустим, мы хотим подсчитать общую стоимость всех элементов этого списка. Мы можем сделать это, например, вот так:

[Скачать `shop1.erl`](#)

```
-module(shop1).
-export([total/1]).
total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([]) -> 0.
```

Давайте поэкспериментируем с этим:

```
2> c(shop1).
{ok,shop1}
```

```
3> shop1:total([]).  
0
```

Почему здесь 0? А потому, что вторым вариантом применения в функции `total/1` является `total([]) -> 0;`

```
4> shop1:total([{milk,3}]).  
21
```

Вызов функции `total([{milk,3}])` подошел клаузе `total([{What,N}|T])` в которой `T = []` (поскольку `[X]=[X|[]]`). После успешного сопоставления, переменные функции принимают следующие значения `{What -> milk, N -> 3, T -> []}`. Затем выполняется тело клаузы функции (`shop:cost(What) * N + total(T)`). Все переменные заменяются на присвоенные им значения. Таким образом, тело функции выглядит как `shop:cost(milk) * 3 + total([])`.

`shop:cost(milk)` равно 7, а `total([])` равно 0; следовательно, значение этого выражения равно `7*3+0 = 21`.

Когда мне ставить точку с запятой?

В Эрланге мы используем три типа пунктуации:

- Запятые (,) разделяют аргументы в вызовах функции, конструкторах данных и шаблонах.
- Точки (.) (с последующим пробелом) разделяют функции и выражения в оболочке Эрланга.
- Точка с запятой (;) разделяет клаузы, которые мы используем в различных контекстах: в объявлении функций, а так же в блоках `case`, `if`, `try..catch` и в выражениях `receive`.

Всякий раз когда мы имеем набор шаблонов, с последующими за ними выражениями, мы используем точку с запятой как их разделитель:

```
Pattern1 ->  
    Expressions1;  
Pattern2 ->  
    Expressions2;  
    ...
```

Как насчет более сложных аргументов?

```
5> shop1:total([{pears,6},{milk,3}]).
```

На этот раз в первой клаузе переменные связываются со следующими значениями:

`{What -> pears, N -> 6, T -> [{milk,3}]}.` В результате: `shop:cost(pears) * 6 + total([{milk,3}])`, или же `9 * 6 + total([{milk,3}])`.

Но мы уже знаем, что результатом работы `total([{milk,3}])` будет 21, поэтому результат равен $9 \cdot 6 + 21 = 75$.

И, наконец:

```
6> shop1:total(Buy).
123
```

Перед тем как закончить этот раздел, давайте более детально рассмотрим функцию `total`. Функция `total(L)` работает в зависимости от результата анализа аргумента `L`. Тут возможно два варианта: или `L` - это не пустой список, или же наоборот. Мы реализовали клаузы для каждого возможного варианта, вот так:

```
total([Head|Tail]) ->
    some_function_of(Head) + total(Tail);
total([]) ->
    0.
```

В нашем случае, `Head` была шаблоном `{What,N}`. Первая клауза срабатывает в случае, когда в нее передается непустой список. Она отделяет от него первый элемент ("голову"), что-то с ним делает, и потом функция вызывает саму себя для оставшейся части списка ("хвоста"). Вторая клауза срабатывает, когда полученный список пуст (`[]`).

На самом деле, функция `total/1` делает две разные вещи. Во-первых, она находит цену каждого элемента в списке, и, потом, складывает все их значения. Мы можем переписать функцию `total` таким образом, что бы разделить получение значения каждого элемента и сложение этих значений. Получившийся код будет более ясным и простым для понимания. Чтобы сделать это, мы напишем две небольшие функции для работы с списками и назовем их `sum` и `map`. Но до того, как мы поговорим об этом, давайте немного познакомимся с анонимными функциями `funs`. После этого мы напишем наши функции `sum` и `map`, а затем улучшим нашу функцию `total`.

3.3. Функции с одинаковыми именами и различной арности (arity)

Арность функции - это количество аргументов, принимаемых этой функцией. В Эрланге, две функции, объявленные в одном модуле, с одним именем, но разным количеством аргументов, представляют собой две различные функции. Они не имеют ничего общего, кроме как одинакового имени.

По общепринятым соглашениям, Erlang программисты часто используют функции с одинаковыми именами, но разным количеством аргументов, как вспомогательные функции. Например:

Скачать [lib_misc.erl](#)

```
sum(L) -> sum(L, 0).

sum([], N) -> N;
sum([H|T], N) -> sum(T, H+N).
```

Функция `sum(L)` складывает элементы списка `L`. Она делает это используя вспомогательную функцию `sum/2`, хотя, она могла бы быть названа как угодно иначе. Вы можете назвать вспомогательную функцию `hedgehog/2`, и принцип работы программы не изменится. Но `sum/2` это более лучший вариант для имени такой функции, так как он дает читателю вашей программы подсказку о ее содержании, а так же избавляет от необходимости придумывать новое имя для функции (что, как известно, не просто).

3.4. Анонимные функции(Funs)

`funs` ("фаны") это "анонимные" функции. Они называются так, потому что у них нет имени. Давайте немного поэкспериментируем. Для начала объявим функцию и сопоставим ее с переменной `Z`:

```
1> Z = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

Когда мы объявляем анонимную функцию, оболочка Erlang выводит `#Fun<...>`, где `...` это какое-то странное число. Но здесь это не важно. Сейчас мы можем сделать с анонимными функциями только одну вещь, а именно, применить ее к списку аргументов, например:

```
2> Z(2).
4
```

`Z` не самое лучшее имя для функции; более лучшим вариантом будет имя `Double`, так

как оно описывает то, что функция делает:

```
3> Double = Z.  
#Fun<erl_eval.6.10732646>  
4> Double(4).  
8
```

Анонимная функция может принимать любое количество аргументов. Мы можем написать функцию вычисления гипотенузы прямоугольного треугольника следующим образом:

```
5> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.  
#Fun<erl_eval.12.115169474>  
6> Hypot(3,4).  
5.00000
```

Если количество аргументов будет неправильным, вы получите сообщение об ошибке:

```
7> Hypot(3).  
exited: {{badarity,{#Fun<erl_eval.12.115169474>,[3]}},  
[{erl_eval,expr,3}]}  
8>
```

Почему эта ошибка называется `badarity`? Вспомните, что арностью называется количество аргументов принимаемых функцией. `badarity` означает, что Erlang не может найти функцию с передаваемым именем, которая принимает переданное количество параметров - наша функция принимает 2 аргумента, а мы передаем только один.

Анонимная функция может также иметь различные варианты применения (клаузы). Приведем функцию которая может конвертировать значения температуры из шкалы Фаренгейта в шкалу Цельсия и наоборот:

```
8> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};  
8> ({f,F}) -> {c, (F-32)*5/9}  
8> end.  
#Fun<erl_eval.6.56006484>  
9> TempConvert({c,100}).  
{f,212.000}  
10> TempConvert({f,212}).  
{c,100.000}  
11> TempConvert({c,0}).  
{f,32.0000}
```

Обратите внимание: Выражение в строке 8 занимается несколько строк. После того,

как мы начали вводить выражение, оболочка повторяет приглашение "8>" каждый раз как мы вводим новую строку. Это значит, что выражение не закончено, и оболочка ожидает продолжение ввода.

Эрланг - это функциональный язык программирования. Кроме всего прочего это означает, что анонимные функции могут быть переданы как аргументы для функции, а также, что функции (или анонимные функции) могут возвращать анонимные функции в качестве результата.

Функция, которая возвращает другие функции, или же может принимать другие функции в качестве своих аргументов, называется *функцией высшего порядка*. Мы увидим несколько примеров таких функций в следующих разделах.

Все это пока может звучать не так восхитительно, поскольку мы еще не видели, что можно делать с анонимными функциями. Пока код анонимных функций выглядел точно также, как и код обычных функций в модуле, но, как правило, это не так. Функции высшего порядка это очень существенная часть функциональных языков программирования - они просто зажигают огонь в теле кода. И когда вы научитесь ими пользоваться - вы полюбите их. В будущем мы встретимся с ними в огромном количестве.

Функции, принимающие другие функции в качестве аргументов

Модуль `lists`, входящий в стандартные библиотеки Эрланга, экспортирует несколько функций, которые принимают другие функции в качестве аргументов. Наиболее полезная из них это функция `lists:map(F, L)`. Она возвращает список, созданный применением функции F к каждому элементу из списка L:

```
12> L = [1,2,3,4].  
[1,2,3,4]  
13> lists:map(Double, L).  
[2,4,6,8].
```

Другая полезная функция - `lists:filter(P, L)`, она возвращает новый список из таких элементов E списка L, для которых функция P(E) равна true.

Давайте создадим функцию `Even(X)`, которая возвращает true, если X - это четное число:

```
14> Even = fun(X) -> (X rem 2) =:= 0 end.  
#Fun<erl_eval.6.56006484>
```

Здесь `X rem 2` вычисляет остаток от деления `X` на `2`, а оператор `=:=` сравнивает его с нулем. Теперь мы можем использовать функцию `Even` как аргумент функции

`map` и `filter`:

```
15> Even(8).  
true  
16> Even(7).  
false  
17> lists:map(Even, [1,2,3,4,5,6,8]).  
[false,true,false,true,false,true,true]  
18> lists:filter(Even, [1,2,3,4,5,6,8]).  
[2,4,6,8]
```

Мы называем функции, такие как `map` и `filter`, которые делают что-либо со списком за один вызов функции, список-за-раз (*list-at-a-time*) операциями. Использование таких операций делает наши программы более короткими и простыми для понимания. И более простыми они становятся потому, что мы можем обработать каждый элемент списка за один логический шаг нашей программы. Иначе нам пришлось бы считать элементарным шагом нашей программы каждую индивидуальную операцию, над каждым элементом списка.

Функции, возвращающие функции

Функции могут использоваться не только в качестве аргументов других функций (таких как `map` и `filter`). Функции могут также возвращаться другими функциями.

Приведем пример - допустим, что у нас есть список чего-либо, предположим фруктов:

```
1> Fruit = [apple,pear,orange].  
[apple,pear,orange]
```

Теперь объявим функцию `MakeTest(L)`, которая преобразует любой список `L` в функцию, которая проверяет, находится ли ее аргумент в этом списке `L`:

```
2> MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.  
#Fun<erl_eval.6.56006484>  
3> IsFruit = MakeTest(Fruit).  
#Fun<erl_eval.6.56006484>
```

`lists:member(X, L)` возвращает `true` если `X` находится в списке `L`, в противном случае она возвращает `false`. Давайте протестируем нашу функцию:

```
4> IsFruit(pear).  
true  
5> IsFruit(apple).  
true
```

```
6> IsFruit(dog).  
false
```

Также мы можем использовать ее как аргумент функции `lists:filter/2`:

```
7> lists:filter(IsFruit, [dog,orange,cat,apple,bear]).  
[orange,apple]
```

Описания анонимных функций, которые сами возвращают анонимные функции, требуют некоторого времени, чтобы освоиться с ними. Так что давайте немного отделим, для ясности, нотацию описаний от того, что происходит на самом деле.

Функции, возвращающие "нормальные" значения, выглядят следующим образом:

```
1> Double = fun(X) -> ( 2 * X ) end.  
#Fun<erl_eval.6.56006484>  
2> Double(5).  
10
```

Код в круглых скобках `(2 * X)` это фактически "возвращаемое значение" функции.

Теперь давайте попробуем поместить анонимную функцию внутрь круглых скобок.

Помним, что выражение внутри скобок - это возвращаемое значение:

```
3> Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.  
#Fun<erl_eval.6.56006484>
```

Функция `fun(X) -> X * Times end` внутри скобок - это просто функция от `X`, но как в ней появилась переменная `Times`? Ответ: это просто аргумент "внешней" анонимной функции.

Вызов `Mult(3)` вернет `fun(X) -> X * 3 end`, что является телом внутренней (обычной) функции, в которой переменная `Times` заменена на число 3. Давайте это протестируем:

```
4> Triple = Mult(3).  
#Fun<erl_eval.6.56006484>  
5> Triple(5).  
15
```

Таким образом, функция `Mult` это обобщение (*generalization*) функции `Double`. Она, вместо вычисления значения, возвращает функцию, которая вычисляет требуемое значение.

Объявление собственных управляемых абстракций

Подождите секундочку - вы заметили это? До сих пор нам так и не попалось никаких выражений с `if`, `switch`, `for` или `while` и это казалось нам совершенно нормальным. Все было написано с использованием сопоставления шаблонам и с помощью функций высшего порядка. До сих пор нам так и не потребовались другие управляющие программные структуры.

Ну, а если нам потребуются дополнительные управляющие структуры, то у нас есть супер-мощное средство для их создания. Давайте рассмотрим пример, как это делается: в Эрланге не существует (совсем) цикла `for`, так давайте сделаем его:

[Скачать `lib_misc.erl`](#)

```
for(Max, Max, F) -> [F(Max)];  
for(I, Max, F) -> [F(I)|for(I+1, Max, F)].
```

Здесь, например, вычисление `for(1,10,F)` создаст список `[F(1), F(2), ..., F(10)]`.

Как же работает сопоставление по шаблону в этом цикле `for`? Первый вариант применения `for` срабатывает только тогда, когда первый и второй аргументы `for` одинаковы. Поэтому, если мы вызовем `for(10,10,F)` то переменной `Max` сопоставится `10`, а результатом этого вызова будет `[F(10)]`. Если мы вызовем `for(1,10,F)`, первый вариант применения `for` не подойдет, так как `Max` не может быть одновременно и `1` и `10`, а второй вариант применения `for` сопоставит `I -> 1` и `Max -> 10`. А значением функции тогда будет `[F(I)|for(I+1,10,F)]`, где `I` заменяется на `1`, а `Max` заменяется на `10`, что составит `[F(1)|for(2,10,F)]`.

Таким образом, у нас появился простой цикл `for`. (Это не совсем тоже самое, что `for` в императивных языках программирования, но для наших целей его достаточно.) Мы можем использовать его для создания списка целых чисел от `1` до `10`:

```
1> lib_misc:for(1,10,fun(I) -> I end).  
[1,2,3,4,5,6,7,8,9,10]
```

Или мы можем его использовать для вычисления квадратов целых от `1` до `10`:

```
2> lib_misc:for(1,10,fun(I) -> I*I end).  
[1,4,9,16,25,36,49,64,81,100]
```

Когда используются функции высшего порядка?

Как мы видели, используя функции высшего порядка, мы можем создавать наши собственные, новые управляющие абстракции, можем передавать функции в

качестве аргументов и мы можем создавать функции, возвращающие абстрактные функции. Но не все эти возможности используются на практике достаточно часто:

- Практически все модули, которые я написал, использовали функции типа `list:map/2` настолько часто, что я уже практически считаю функцию `map` просто частью языка Эрланг. Обращение к функциям `map`, `filter` и `partition` из модуля `lists` встречается очень часто.
- Иногда я создавал свои собственные управляющие абстракции. Хотя это встречается гораздо реже, чем вызов высших функций из стандартных библиотечных модулей. Такое бывает до нескольких раз, для большого модуля.
- Написание функций, которые возвращают абстрактные функции я использовал крайне редко. Если бы я писал сотню модулей, то эта техника встречалась бы скорее всего, только в одном или двух из них. Программы с функциями, возвращающими абстрактные функции бывает весьма трудно отлаживать. Но, с другой стороны, мы можем использовать такие функции для реализации таких вещей, как ленивые вычисления, и мы можем легко создавать возвратные парсеры и комбинаторы парсеров, являющиеся функциями, которые возвращают парсеры. (примечание переводчика: Ж8-О)

По мере накопления вашего опыта, вы можете обнаружить, что возможность создавать свои собственные управляющие структуры может феноменально сократить размер ваших программ и, иногда, сделать их гораздо понятнее. Поскольку теперь вы можете создавать управляющие структуры конкретно под решение ваших проблем и поскольку теперь вы не ограничены небольшим и фиксированным набором управляющих структур вашего языка программирования.

Стандартные ошибки

Некоторые читатели, часто пытаются, по ошибке, набрать примеры, приводимые в листингах модулей в этой книге, в оболочке Эрланга. Зря, это вовсе не команды для оболочки. Вы получите за это очень странные сообщения об ошибках. Поэтому мы вас предупреждаем: не делайте этого.

Если вы случайно выберете имя для своего модуля, которое совпадает с одним из системных модулей, тогда, после компиляции вашего модуля, вы получите странное сообщение, утверждающее, что вы не можете загрузить модуль из такой-то директории. Просто переименуйте ваш модуль (не забудьте про директиву `-module(...)` внутри файла модуля) и удалите все `beam` файлы, которые вы могли создать при компиляции вашего первоначального модуля.

3.5. Простая обработка списков

Теперь, когда мы познакомились с анонимными функциями, мы можем вернуться к написанию `sum` и `map`, которые нам потребуются для улучшения `total` (о которой, я уверен, вы еще не забыли!).

Мы начнем с функции `sum`, которая вычисляет сумму элементов в списке:

Скачать [mylists.erl](#)

```
sum([H|T]) -> H + sum(T);  
sum([]) -> 0.
```

Отметим, что, в данном случае, порядок написания двух клауз данной функции не важен, поскольку, первая клауза срабатывает только на непустые списки, а вторая только на пустой список. То есть они - взаимно исключающие.

Теперь мы можем протестировать функцию `sum`:

```
1> c(mylists).%% <-- Это в последний раз  
{ok, mylists}  
2> L = [1,3,10].  
[1,3,10]  
3> mylists:sum(L).  
14
```

В строке 1 я скомпилировал модуль `lists`. Но с этого момента, я буду, обычно, не показывать команду компиляции модуля, и вам нужно будет не забывать делать это самостоятельно.

Далее - все просто. Давайте рассмотрим выполнение `sum` по шагам:

1. `sum([1,3,10])`
2. `sum([1,3,10]) = 1 + sum([3,10]) (by 1)`
3. `= 1 + 3 + sum([10]) (by 1)`
4. `= 1 + 3 + 10 + sum([]) (by 1)`
5. `= 1 + 3 + 10 + 0 (by 2)`
6. `= 14`

И, наконец, давайте посмотрим `map/2`, с которой мы уже встречались ранее. Вот как она определяется:

Скачать [mylists.erl](#)

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F, T)].
```

1 строка - Первая клауза этой функции говорит нам о том, что она делает с пустым списком. Применение любой функции к пустому списку (в котором ничего нет!) дает нам снова пустой список.

2-я строка - Вторая клауза описывает, что происходит с непустым списком, состоящим из головы `H` и хвоста `T`. Это тоже просто. Получается новый список с головой `F(H)` и хвостом `map(F, H)`.

Примечание: Данное определение функции `map/2` скопировано нами из модуля `lists` стандартной библиотеки в модуль `mylists`. С модулем `mylists` вы можете делать все, что угодно. Но ни при каких обстоятельствах не пытайтесь создать свой собственный модуль с именем `lists`, если только вы абсолютно уверены в том, что вы делаете.

Мы можем поработать с `map`, используя парочку функций, которые удваивают или возводят в квадрат элементы из списка, следующим образом:

```
1> L = [1,2,3,4,5].
[1,2,3,4,5].
2> mylists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
3> mylists:map(fun(X) -> X*X end, L).
[1,4,9,16,25]
```

И на этом все, о функции `map`? Ну, на самом деле, не совсем. Позднее, мы покажем ее еще более короткую версию, написанную с использованием обработчиков списков. А в разделе 20.2 *Распараллеливание последовательного кода*, мы покажем, как можно вычислять все элементы получаемого после `map` списка *параллельно* (что ускорит выполнение нашей программы на многоядерном компьютере), но это мы сейчас забежали очень и очень далеко вперед. Сейчас, мы лишь можем только переписать функцию `total` с помощью этих двух функций:

[Скачать shop2.erl](#)

```
-module(shop2).
-export([total/1]).
-import(lists, [map/2, sum/1]).
total(L) ->
sum(map(fun({What, N}) -> shop:cost(What) * N end, L)).
```

Мы можем проверить, как она работает на следующем примере:

```
1> Buy =
[ {oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3} ].
[ {oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3} ]
2> L1=lists:map(fun({What,N}) -> shop:cost(What) * N end, Buy).
[20,8,20,54,21]
3> lists:sum(L1).
123
```

Как я пишу свои программы

Когда я пишу программу, то я пользуюсь подходом: "напиши чуть-чуть - потом протестируй это". Я начинаю с маленького модуля, состоящего всего из нескольких функций, потом я компилирую его и тестирую с помощью нескольких команд в оболочке Эрланга. Когда он меня вполне удовлетворяет, я дописываю в нем еще несколько функций, компилирую его и тестирую. И так далее.

Часто я не представляю себе заранее, какие именно структуры данных мне потребуются в моей программе, но по мере того, как я попробую несколько простых примеров, я уже могу оценить, какие подходящие варианты из них мне следует выбрать.

Я предпочитаю, скорее "наращивать" свои программы, а не продумывать их заранее полностью, до их написания. При таком способе, я, скорее всего, не сделаю заранее крупной ошибки, до того, как обнаружу, что все пошло не так. И, кроме того, так гораздо веселее. Я немедленно получаю обратную связь и вижу, что мои идеи начинают работать, как только я реализовал их в программе.

А когда я понимаю, как что-то можно сделать в командной оболочке, то, часто я иду и пишу для данного случая make-файл и, возможно, немного кода, которые воспроизводят то, что я научился делать в оболочке Эрланга.

Обратите, также, внимание на использование объявлений `-import` и `-export` в этом модуле:

Объявление `-import(lists, [map/2, sum/1])`. означает, что функции `map/2` и `sum/1` импортируются из модуля `lists`. Это означает, что мы можем писать `map(Fun, ...)` вместо `lists:map(Fun, ...)`. А поскольку функция `cost/1` не была указана в объявлении `-import`, то нам придется использовать ее "полное" имя `shop:cost`.

Объявление `-export([total/1])` означает, что функция `total/1` может вызываться снаружи данного модуля `shop2`. Только проэкспортированные так функции могут

вызываться снаружи модулей.

Возможно, сейчас вам кажется, что нашу функцию `total` больше уже улучшить нельзя, но вы не правы. Это вполне возможно. Для этого мы используем обработчики списков (list comprehension).

3.6. Обработчики списков

Обработчики списков - это выражения, которые создают списки без использования анонимных функций, отображений (maps) или фильтров. Это делает нашу программу еще проще и доступнее для понимания.

Мы начнем с небольшого примера. Предположим, что у нас имеется список `L`:

```
1> L = [1,2,3,4,5].  
[1,2,3,4,5]
```

И предположим, что мы хотим удвоить каждый элемент в данном списке. Мы это уже делали раньше, но я вам напомню:

```
2>lists:map(fun(X) -> 2*X end, L).  
[2,4,6,8,10]
```

Но существует гораздо более легкий способ сделать это, используя обработчики списков:

```
4> [2*X || X <- L ].  
[2,4,6,8,10]
```

Такая запись вида `[F(X) || X <- L]` означает "список из элементов `F(X)`, где `X` берутся из списка `L`". Следовательно, запись `[2*X || X <- L]` означает "список значений `2*X`, где `X` берутся из списка `L`".

Чтобы увидеть, как пользоваться обработчиками списков, мы можем ввести несколько команд в оболочку Эрланга и посмотреть, что из этого получится: Мы начнем с определения списка `Buy` (покупки):

```
1> Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].  
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
```

А теперь, давайте удвоим число покупок по каждой из позиций из исходного списка:

```
2> [{Name, 2*Number} || {Name, Number} <- Buy].  
[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]
```

Отметим дополнительно, что набор (кортеж) {Name, Number} с права от знака `||` - это образец (паттерн), которому сопоставляется каждый элемент из списка `Buy`. А набор (кортеж) слева от знака `||` - `Name, 2*Number}` - называется конструктором.

Предположим теперь, что мы хотим посчитать общую стоимость всех элементов в исходном нашем списке. Это можно сделать следующим образом:

Вначале, заменим имя каждой из покупок на ее цену:

```
3> [{shop:cost(A), B} || {A, B} <- Buy].  
[{5,4},{8,1},{2,10},{9,6},{7,3}]
```

Теперь перемножим эти значения:

```
4> [shop:cost(A) * B || {A, B} <- Buy].  
[20,8,20,54,21]
```

А теперь, просуммируем их:

```
5> lists:sum([shop:cost(A) * B || {A, B} <- Buy]).  
123
```

И, наконец, если мы хотим превратить все это в одну функцию, мы можем написать следующее:

```
total(L) ->  
lists:sum([shop:cost(A) * B || {A, B} <- L]).
```

Таким образом, вы видите, что обработчики списков действительно могут сделать ваш код очень коротким и понятным. Мы даже можем с их помощью, чисто ради развлечения, дать более короткое определение функции `map`:

```
map(F, L) -> [F(X) || X <- L].
```

В самом общем виде выражение для обработчика списков выглядит следующим образом:

```
[X || Qualifier1, Qualifier2, ...]
```

где `X` - это произвольное выражение, а каждый из определителей (Qualifier) это либо

генератор, либо фильтр. При этом:

Генераторы записываются в виде Образец <- Списочное Выражение, где
Списочное Выражение - это любое выражение, дающее на выходе список термов Эрланга.

Фильтры - это либо предикаты (функции, которые возвращают либо true (истина) либо false (ложь)), либо же просто логические выражения.

Заметьте, что генератор в обработчике списков тоже работает, как своеобразный фильтр. Например:

```
1> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4}], hello, "wow" ] ].  
[1,4]
```

Мы закончим этот раздел, посвященный обработчикам списков, несколькими небольшими примерами:

Быстрая сортировка

Вот как можно написать алгоритм сортировки, используя всего лишь два обработчика списков:

Скачать [lib_misc.erl](#)

```
qsort([]) -> [];  
qsort([Pivot|T]) ->  
  qsort([X || X <- T, X < Pivot])  
  ++ [Pivot] ++  
  qsort([X || X <- T, X >= Pivot]).
```

Здесь `++` - это инфиксный оператор добавления. А Pivot переводится как "центр вращения".

(Данный код приведен здесь, скорее, из-за своей элегантности, чем своей эффективности. Использование оператора `++` таким образом, вообще-то, не считается хорошей практикой программирования.)

Пример:

```
1> L=[23,6,2,9,27,400,78,45,61,82,14].  
[23,6,2,9,27,400,78,45,61,82,14]  
2> lib_misc:qsort(L).  
[2,6,9,14,23,27,45,61,78,82,400]
```

Чтобы понять, как работает эта функция мы рассмотрим это все по шагам:

Для начала у нас есть список `L` и мы вызываем `qsort(L)`. Срабатывает вторая клауза функции `qsort`:

```
3> [Pivot|T] = L.  
[23,6,2,9,27,400,78,45,61,82,14]
```

которая связывает переменные `Pivot` -> 23 и `T` -> [6,2,9,27,400,78,45,61,82,14].

Теперь мы разделяем список `T` на два списка : один из элементов, которые меньше чем `Pivot`, а второй - из элементов, которые больше или равны `Pivot`:

```
4> Smaller = [X || X <- T, X < Pivot].  
[6,2,9,14]  
5> Bigger = [X || X <- T, X >= Pivot].  
[27,400,78,45,61,82]
```

Теперь мы можем отсортировать списки `Smaller` и `Bigger` и соединить их обратно с `Pivot`:

```
qsort( [6,2,9,14] ) ++ [23] ++ qsort( [27,400,78,45,61,82] )  
= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]  
= [2,6,9,14,23,27,45,61,78,82,400]
```

Тройки Пифагора

Тройки Пифагора - это такие наборы из трех натуральных чисел `{A,B,C}` для которых верно равенство: $A^2 + B^2 = C^2$.

Нижеследующая функция `pythag(N)` генерирует список всех натуральных чисел `{A,B,C}`, для которых выполняется указанное равенство и сумма которых меньше или равна `N`:

[Скачать lib_misc.erl](#)

```
pythag(N) ->  
  [ {A,B,C} ||  
    A <- lists:seq(1,N),  
    B <- lists:seq(1,N),  
    C <- lists:seq(1,N),  
    A + B + C =< N,  
    A * A + B * B =:= C * C  
  ].
```

Небольшие пояснения: функция `lists:seq(1, N)` возвращает нам список из всех целых чисел от 1 до N. Следовательно, запись `A <- lists:seq(1, N)` означает, что A принимает все значения от 1 до N. И, значит, нашу программу можно прочитать следующим образом: "Возьми все значения A от 1 до N, все значения B от 1 до N и все значения C от 1 до N? такие что A+B+C мене или равно N и $A^A + B^B = C^C$."

```
1> lib_misc:pythag(16).
[{3,4,5},{4,3,5}]
2> lib_misc:pythag(30).
[{3,4,5},{4,3,5},{5,12,13},{6,8,10},{8,6,10},{12,5,13}]
```

Анаграммы

Если вы интересуетесь традиционными Английскими словарными играми, то вы несомненно знакомы с разгадыванием анаграмм. Давайте используем Эрланг для нахождения всех перестановок строки символов, с помощью симпатичной, маленькой функции `perms` следующего вида:

Скачать [lib_misc.erl](#)

```
perms([]) -> [[]];
perms(L) -> [[H|T] || H <- L, T <- perms(L--[H])].
```

Здесь `--` это оператор вычита списка из другого. Он вычитает все элементы второго списка из первого. Его более точное определение дано в разделе 5.4 *Операции со списками ++ и --.*

```
1> lib_misc:perms("123").
["123","132","213","231","312","321"]
2> lib_misc:perms("cats").
["cats", "cast", "ctas", "ctsa", "csat", "csta", "acts", "acst",
 "atcs", "atsc", "asct", "astc", "tcas", "tcsa", "tacs", "tasc",
 "tsca", "tsac", "scat", "scta", "sact", "satc", "stca", "stac"]
```

Я не буду объяснять как работает функция `perm`, поскольку это объяснение получится гораздо длиннее, чем сама запись этой функции. Так что, вы можете разобраться с этим сами! (Но я дам Вам намек: чтобы вычислить все перестановки X123, вам надо вычислить все перестановки 123 (что составит [123, 132, 213, 231, 312, 321], а теперь вам надо вставить X во все возможные места каждой из полученных перестановок. Например, добавление X к 123 даст нам X123 1X23 12X3 123X; добавление X к 132 даст нам X132 1X32 13X2 132X и так далее. Примените это правило рекурсивно.)

3.7. Арифметические выражения

Все возможные арифметические выражения в Эрланге приведены ниже в таблице 3.1. В ней каждый арифметический оператор имеет один - два аргумента, которые могут быть либо целым, либо числом (что означает либо целое, либо вещественное число).

Также, с каждым оператором ассоциирован его *приоритет*, от которого зависит порядок выполнения сложных арифметических выражений: сначала выполняются все операторы с приоритетом 1 (слева направо), потом - с приоритетом 2 и так далее.

Оператор	Описание	Типы аргументов	Приоритет
<code>+X</code>	<code>+X</code>	Число	1
<code>-X</code>	<code>-X</code>	Число	1
<code>X * Y</code>	<code>X * Y</code>	Число	2
<code>X / Y</code>	<code>X / Y</code> (деление с дробной частью)	Число	2
<code>bnot X</code>	побитовое отрицание X	Целое	2
<code>X div Y</code>	целочисленное деление X на Y	Целое	2
<code>X rem Y</code>	целый остаток от деления X на Y	Целое	2
<code>X band Y</code>	побитовое И между X и Y	Целое	2
<code>X + Y</code>	<code>X + Y</code>	Число	3
<code>X - Y</code>	<code>X - Y</code>	Число	3
<code>X bor Y</code>	побитовое ИЛИ между X и Y	Целое	3
<code>X bxor Y</code>	побитовое исключающее ИЛИ	Целое	3
<code>X bsl N</code>	Арифметический сдвиг битов X влево на N бит	Целое	3
<code>X bsr N</code>	Арифметический сдвиг битов X вправо на N бит	Целое	3

Таблица 3.1 : Арифметические выражения

Для изменения порядка выполнения арифметических операций можно использовать круглые скобки. Операторы с одинаковым приоритетом считаются лево-ассоциативными и выполняются слева направо.

3.8. Контролеры (Guards)

Контролеры позволяют нам еще более усилить мощь механизма сопоставления по образцу. (прим. переводчика: "Контроллеры еще также иногда называют охранными выражениями") Используя их, мы можем дополнительно проверять и сравнивать переменные из образца. Предположим, что мы хотим написать функцию `max(X, Y)`, которая должна вычислять максимум из `X` и `Y`. С помощью контролеров, мы можем написать ее следующим образом:

```
max(X, Y) when X > Y -> X;  
max(X, Y) -> Y.
```

Первая клауда этой функции срабатывает когда `X` больше `Y` и возвращает нам `X`.

А, если первая клауда не срабатывает, то в дело вступает вторая клауда, которая нам просто всегда возвращает `Y`. При этом `Y` должен быть больше или равно `X`, поскольку, иначе, сработала бы первая клауда этой функции.

Вы можете использовать контролеры и при определении функции, в их заголовках, где они следуют за ключевым словом `when`. Либо же вы можете их использовать в любом месте программы, где можно использовать выражение. И если контролеры используются как выражение, то они вычисляются до одного из атомов: `true` или `false`. Считается, что если вычисление контролера привело к `true`, то его вычисление прошло успешно. В противном случае - неудачным.

Последовательности контролеров

Последовательностью контролеров называют либо одиночного контролера, либо последовательность контролеров, разделенных точкой с запятой `;`. Значение последовательности контролеров `G1; G2; ...; Gn` считается равным `true` (истина), если хотя бы один из контролеров `G1, G2, ...` принимает значение `true`.

А теперь: контролером в Эрланге называют последовательность контрольных выражений, разделенных запятыми. Последовательность контрольных выражений считается истинной тогда и только тогда, когда все истинны все контрольные выражения входящие в нее.

Множество выражений, которые могут входить в контрольные выражения, несколько меньше, чем все доступные выражения в Эрланге. И причиной такого ограничения для контрольных выражений является необходимость гарантировать, что их вычисление будет абсолютно не иметь побочных эффектов. Контролеры являются частью выражений сопоставления по образцу и, поскольку, оно не имеет побочных эффектов, они нам также не нужны и при вычислении контрольных выражений.

Кроме того, контролеры не могут быть логическими выражениями, определяемыми пользователями, поскольку нам опять таки нужны гарантии, что в них не будет побочных эффектов и, что они завершат свое выполнение.

Следующие синтаксические формы языка Эрланг допустимы в контрольных выражениях:

- атом true (истина)
- Другие константы (термы и связанные переменные) все они заменяются на false (ложь) в контрольных выражениях.
- Вызовы контрольных предикатов из нижеследующей Таблицы 3.2 и встроенные функции (BIF) из Таблицы 3.3.
- Сравнения термов (Таблица 5.3)
- Арифметические выражения (Таблица 3.1, ранее)
- Логические выражения (Раздел 5.4 *Логические выражения*)
- Упрощенные (или укороченные) логические выражения (Раздел 5.4. подраздел *Упрощенные логические выражения*)

При вычислении контрольных выражений выполняются правила старшинства операторов, описанные в разделе 5.4, подразделе *Старшинство операторов*.

Примеры контролеров

```
f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...
```

Эта запись означает, что "когда X это целое число и X больше чем Y и Y меньше чем 6, то ...". Запятая, разделяющая отдельные проверки в контролере означает логическое "И".

```
is_tuple(T), size(T) =:= 6, abs(element(3, T)) > 5  
element(4, X) =:= hd(L)  
...
```

Первый контролер означает, что T - это кортеж из шести элементов и абсолютное значение третьего элемента кортежа больше 5. Вторая строка означает, что 4-й элемент кортежа X идентичен первому элементу списка L.

```
X =:= dog; X =:= cat  
is_integer(X), X > Y ; abs(Y) < 23  
...
```

Первый контролер проверяет, что X это либо cat (кошка), либо `dog (собака). Второй

контролер проверяет, что, либо X - это целое число, которое больше чем Y; либо, что абсолютное значение Y меньше чем 23.

А вот несколько примеров контролеров использующих упрощенные (или укороченные) логические выражения:

```
A >= -1.0 andalso A+1 > B  
is_atom(L) orelse (is_list(L) andalso length(L) > 2)
```

Для опытных: Общей причиной разрешения использовать логические выражения в контролерах, было стремление сделать их синтаксически похожими на все прочие выражения. А причина для введения операторов orelse и andalso состоит в том, что операторы and/or изначально были определены через обязательное вычисление обоих своих аргументов. Однако, в контролерах может быть разница при использовании and и andalso, а также or и orelse. Например, рассмотрим следующие контролеры:

```
f(X) when (X == 0) or (1/X > 2) ->  
...  
g(X) when (X == 0) orelse (1/X > 2) ->  
...
```

Контролер для `f(x)` выдаст ошибку, когда `X` будет нулем, но сработает для `g(x)`.

Но на практике, весьма малое количество программ использует сложные контролеры, а для решения большинства проблем, вполне хватает простых `,` контролеров.

Предикат	Его значение
<code>is_atom(X)</code>	X - это атом
<code>is_binary(X)</code>	X - бинарная последовательность
<code>is_constant(X)</code>	X - константа
<code>is_float(X)</code>	X - это действительное число
<code>is_function(X)</code>	X - это функция
<code>is_function(X, N)</code>	X - это функция с числом аргументов (арностью) равным N
<code>is_integer(X)</code>	X - это целое число
<code>is_list(X)</code>	X - это список
<code>is_number(X)</code>	X - это целое или действительное число
<code>is_pid(X)</code>	X - это идентификатор процесса (PID)
<code>is_port(X)</code>	X - это порт

<code>is_reference(X)</code>	X - это ссылка
<code>is_tuple(X)</code>	X - это кортеж
<code>is_record(X, Tag)</code>	X - это запись с типом Tag
<code>is_record(X, Tag, N)</code>	X - это запись с типом Tag и размера N

Таблица 3.2. Предикаты используемые в контролерах

Использование контролера `true` (истина)

Во-первых, вы можете спросить: а зачем вообще нужен такой контролер `true`? Причина здесь в том, что атом `true` удобно использовать для контролера "все прочие" в конце выражения `if`. Приблизительно вот так:

```
if
  Guard -> Expressions;
  Guard -> Expressions;
  ...
  true -> Expressions
end
```

Само выражение `if` будет обсуждаться нами в разделе 3.10 *Выражение if*.

Устаревшие контрольные выражения

Если вы столкнетесь со старым кодом на Эрланге, написанным несколько лет назад, то вы можете обнаружить, что тогда проверки в контролерах были несколько иными. Тогда в контролерах использовались проверки `atom(X)`, `constant(X)`, `float(X)`, `integer(X)`, `list(X)`, `number(X)`, `pid(X)`, `port(X)`, `reference(X)`, `tuple(X)`, и `binary(X)`. Эти проверки означают тоже самое, что и современные их варианты типа `is_atom(X)` и так далее. Использование старых названий в современном коде не рекомендуется.

Функция	Значение функции
<code>abs(X)</code>	Абсолютное значение X.
<code>element(N, X)</code>	Элемент номер N из кортежа X.
<code>float(X)</code>	Конвертация числа X в действительное число.
<code>hd(X)</code>	Голова списка X.
<code>length(X)</code>	Длина списка X.
<code>node()</code>	Данный узел Эрланга.
<code>node(X)</code>	Узел на котором X был создан, где X это может быть процесс, идентификатор, ссылка или порт.

<code>round(X)</code>	Конвертирует число X в целое число.
<code>self()</code>	Идентификатор этого (данного) процесса.
<code>size(X)</code>	Размер X, где X - это кортеж или бинарная последовательность.
<code>trunc(X)</code>	Обрезает число X до целого (отбрасывает дробную часть).
<code>tl(X)</code>	Хвост списка X.

Таблица 3.3. Встроенные функции для контролеров

3.9. Записи

Когда мы используем в нашей программе кортежи, мы можем столкнуться с трудностями, если число элементов в этих кортежах станет, вдруг, очень большим. Тогда становится достаточно трудно помнить, какой элемент в кортеже что означает. А записи позволяют привязать к каждому элементу кортежа его собственное имя, что решает данную проблему.

В маленьких кортежах таких сложностей обычно не возникает и, поэтому, мы часто видим программы, которые манипулируют небольшими кортежами, не вызывающих сомнений в предназначении своих элементов.

Записи объявляются с помощью следующего синтаксиса:

```
-record(Name, {
    %% the next two keys have default values
    key1 = Default1,
    key2 = Default2,
    ...
    %% The next line is equivalent to
    %% key3 = undefined
    key3,
    ...
}).
```

Предупреждение: `record` это вовсе не команда для оболочки Эрланга (используйте в ней команду `rr`, описанную чуть ниже). Определение записи может быть использовано только в исходном коде Эрланга, но не в его командной оболочке.

В вышеуказанном примере, `Name` - это имя всей этой записи. `key1`, `key2` и так далее - это имена полей этой записи. Все эти имена должны быть атомами Эрланга. При этом, `key1` и `key2` имеют значения по-умолчанию (`Default1` и `Default2`,

соответственно), которые присваиваются этим полям при создании новой записи Name, если для них не указано другого значения. Поле key3 является изначально неопределенным полем записи.

Предположим, например, что мы хотим создать что-то вроде списка дел на будущее. Мы начнем с определения записи todo и сохраним ее в файле. Определения записей могут быть либо сразу включены в файлы с исходным кодом Эрланга, либо помещены в файлы с расширением .hrl и потом включены в файлы с исходным кодом (что является единственным способом, чтобы в разных Эрланг-файлах было одно и тоже определение этих записей).

[Скачать records.hrl](#)

```
-record(todo, {status=reminder, who=joe, text}).
```

Как только запись была определена, мы можем создавать ее представителей в программе.

Чтобы сделать это в командной оболочке Эрланга, нам надо, сначала, прочитать определение записи в оболочку с помощью команды rr (сокращение от read record (прочитать запись)):

```
1> rr("records.hrl").
```

Создание и изменение записей

Теперь мы готовы создавать записи и манипулировать ими:

```
2> X=#todo{}.  
#todo{status = reminder, who = joe, text = undefined}  
3> X1 = #todo{status=urgent, text="Fix errata in book"}.  
#todo{status = urgent, who = joe, text = "Fix errata in book"}  
4> X2 = X1#todo{status=done}.  
#todo{status = done, who = joe, text = "Fix errata in book"}
```

В строках 2 и 3 мы *создали* новые записи. С помощью выражения вида `#todo{key1=Val1, ..., keyN=ValN}` можно создавать новые записи todo, но, при этом все атомы key1, ..., keyN должны быть такими же, как и в определении записи. Если какое-то из полей записи пропущено при ее создании, то этому полю присваивается значение по-умолчанию из определения записи (или undefined, если его там не было).

В строке 4 мы скопировали существующую запись. Синтаксис `X1#todo{status=done}` означает: создать копию записи X1 (которая должна быть типа todo) и изменить в ней

значение поля `status` на `done` (сделано). Обратите внимание, что это будет только копия. Исходная запись при этом не изменится.

Извлечение значений полей из записей

Как и для всего прочего, для этого используется сопоставление по образцу:

```
5> #todo{who=W, text=Txt} = X2.  
#todo{status = done, who = joe, text = "Fix errata in book"}  
6> W.  
joe  
7> Txt.  
"Fix errata in book"
```

Как вы видите, с левой стороны оператора сопоставления по образцу `=` мы написали подобие нашей записи с несвязанными переменными `W` и `Txt`. Если этот оператор завершится успешно (т.е произойдет сопоставление) данные переменные окажутся связанными с соответствующими значениями полей в нашей записи. При этом, если нам надо только одно значение поля записи, то мы можем использовать более простой синтаксис "с точкой":

```
8> X2#todo.text.  
"Fix errata in book"
```

Сопоставление по образцу записей в функциях

Мы можем писать функции, которые сопоставляют по образцу поля записей, либо которые создают новые записи. Для этого, обычно, используется следующий код:

```
clear_status(#todo{status=S, who=W} = R) ->  
% Inside this function S and W are bound to the field  
% values in the record  
%  
% R is the *entire* record  
R\#todo{status=finished}  
% ...
```

(В комментариях тут написано: Внутри этой функции переменные `S` и `W` будут связаны со значениями указанных полей переданной в функцию записи.)

Чтобы сопоставляться с записями определенного, нужного нам типа, мы можем написать определение функции подобно следующему:

```
do_something(X) when is_record(X, todo) ->
```

```
%% ...
```

Данная клауза функции сработает только когда `X` - это запись типа `todo`.

Записи - это замаскированные кортежи

На самом деле записи - это просто кортежи. Давайте заставим оболочку Эрланга забыть про определение записи `todo`:

```
11> X2.  
#todo{status = done,who = joe,text = "Fix errata in book" }  
12> rf(todo).  
ok  
13> X2.  
{todo,done,joe,"Fix errata in book" }
```

В строке 12 мы скомандовали оболочке забыть определение записи `todo`. Поэтому теперь, когда мы пытаемся напечатать `X2`, оболочка показывает ее как кортеж. Внутри программы все представлено только в виде кортежей. Записи - это всего лишь удобная их форма, в которой вы можете дать имена различным элементам кортежа.

3.10. Выражения case и if

До сих пор, мы использовали для решения всех своих задач только механизм сопоставления по образцу. Это делает Эрланг компактным и последовательным. Но, иногда, определять различные клаузы в функциях для каждого случая бывает довольно неудобно. В этом случае мы можем воспользоваться выражениями `case` и `if`.

Выражение case

Выражение `case` имеет следующий синтаксис:

```
case Expression of  
    Pattern1 [when Guard1] -> Expr_seq1;  
    Pattern2 [when Guard2] -> Expr_seq2;  
    ...  
end
```

Оно вычисляется следующим образом. Во-первых вычисляется `Expression`, предположим, что при этом оно принимает значение `Value`. Далее `Value` сопоставляется с `Pattern1` (вместе с опциональным контролером `Guard1`), `Pattern2`

и так далее, до первого успешного сопоставления. Как только это случится, вычисляется соответствующая последовательность выражений (Expr_seqN) и результат этого вычисления становится результатом всего данного case-выражения. Если не один из паттернов не подойдет, то происходит исключительная ситуация.

Ранее мы уже рассматривали функцию `filter(P, L)`. Она возвращает список элементов `X` из списка `L` для которых `P(X)` истинно. Если использовать только сопоставление по образцу, то мы можем определить `filter` следующим образом:

```
filter(P, [H|T]) -> filter1(P(H), H, P, T);
filter(P, []) -> [].
filter1(true, H, P, T) -> [H|filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).
```

Но такое определение довольно некрасиво, так как нам пришлось изобретать еще одну функцию `filter1` и передавать ей все аргументы `filter/2`.

Но мы можем сделать это гораздо более простым образом, используя конструкцию case следующим образом:

```
filter(P, [H|T]) ->
  case P(H) of
    true -> [H|filter(P, T)];
    false -> filter(P, T)
  end;
filter(P, []) -> [].
```

Выражение if

Также в Эрланге имеется и вторая условная конструкция `if`. Вот ее синтаксис:

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
end
```

Она вычисляется следующим образом: Сначала вычисляется контролер `Guard1`. Если его значение равно `true`, то тогда значение всего выражения `if` получается вычислением последовательности выражений `Expr_seq1`. Если же контролер `Guard1` не сработал, то вычисляется `Guard2` и так далее, пока кто-то из них не примет значение Истина (`true`). Если же такого не найдется, то будет поднято исключение.

Часто последним контролером выражения `if` бывает атом `true`, который "пропускает всех", тем самым гарантируя, что, по крайней мере одна из последовательностей выражений будет вычислена, даже если все остальные охранники не сработают.

3.11. Построение списков в естественном порядке

Самым эффективным способом построения списков является добавление новых его элементов в его голову, и, поэтому, мы часто можем увидеть код приблизительно такого вида:

```
some_function([H|T], ..., Result, ...) ->
    H1 = ... H ...,
    some_function(T, ..., [H1|Result], ...);
    some_function([], ..., Result, ...) ->
    {..., Result, ...}.
```

Этот код проходит по списку, берет его голову `H` и вычисляет от нее какое-то значение определяемое в данной функции (мы назвали его `H1`). Потом `H1` добавляется к итоговому списку `Result`.

Когда исходный список закончится, сработает финальная клауда данной функции и итоговая переменная `Result` будет возвращена из данной функции.

Но элементы в списке `Result` будут находиться в обратном порядке к породившем их элементам в исходном списке. Для некоторых ситуаций это вовсе не проблема, но если, все таки, это так, то они легко могут быть переставлены на заключительном шаге.

Основная идея очень проста:

1. Всегда добавляйте новые элементы в голову списка.
2. Когда вы берете исходные элементы из головы исходного списка и добавляете их (или результаты их обработки) в голову результирующего списка, вы получаете список в обратном порядке к исходному.
3. Но, если вам важен правильный порядок следования элементов, используйте функцию `list:reverse/1` которая реализована крайне оптимально (в смысле скорости работы).
4. Избегайте использования других рекомендаций, кроме этих.

Примечание: Когда бы вы не захотели обратить список вспять, вы должны пользоваться функцией `list:reverse` и ничем иным. Если вы захотите посмотреть на ее исходный код в модуле `lists`, то там тоже будет и ее определение. Но учтите, оно

приведено там только для иллюстрации. Компилятор, когда он встречается с вызовом lists:reverse, обращается к гораздо более эффективной внутренней версии этой функции.

Как только вы увидите код, подобный нижеследующему:

```
Lists ++ [H]
```

у вас в голове должен сработать сигнал тревоги, поскольку это очень не эффективный способ, который приемлем, только если список List очень короткий.

3.12. Аккумуляторы

Как нам получить из одной функции два списка? Как нам написать функцию, которая разделяет список целых чисел на два, в которых будут только четные и нечетные числа из исходного списка? Вот один из способов, как это можно сделать:

Скачать [lib.misc.erl](#)

```
odds_and_evens(L) ->
  Odds = [X || X <- L, (X rem 2) =:= 1],
  Evens = [X || X <- L, (X rem 2) =:= 0],
  {Odds, Evens}.
5> lib_misc:odds_and_evens([1,2,3,4,5,6]).
{[1,3,5],[2,4,6]}
```

Но проблема с этим кодом состоит в том, что мы проходим по исходному списку дважды, что не очень страшно, когда он короткий, но если он очень длинный - это может стать проблемой.

Чтобы избежать этого двойного прохождения по списку, мы можем переписать наш код следующим образом:

Скачать [lib.misc.erl](#)

```
odds_and_evens_acc(L) ->
  odds_and_evens_acc(L, [], []).
odds_and_evens_acc([H|T], Odds, Evens) ->
  case (H rem 2) of
    1 -> odds_and_evens_acc(T, [H|Odds], Evens);
    0 -> odds_and_evens_acc(T, Odds, [H|Evens])
  end;
odds_and_evens_acc([], Odds, Evens) ->
```

```
{Odds, Evens}.
```

Теперь мы проходим по списку только один раз и добавляем его четные и нечетные элементы в их собственные выходные списки (которые мы называем аккумуляторами). У этого кода, кроме того, есть еще одно преимущество, которое гораздо менее очевидно: версия с аккумуляторами гораздо более эффективна в смысле использования памяти, чем версия с конструкциями типа `[H || filter(H)]`.

Если мы запустим это, то мы получим почти такой же результат, как и ранее:

```
1> lib_misc:odds_and_evens_acc([1,2,3,4,5,6]).  
{[5,3,1],[6,4,2]}
```

Разница тут в том, что элементы в четном и нечетном списках здесь в обратном порядке. Это является следствием того, как эти списки были получены. Если нам нужны эти списки в порядке следования элементов в исходном списке, нам всего лишь надо сделать реверсирование списка в финальной клаузе нашей функции следующим образом:

```
odds_and_evens_acc([], Odds, Evens) ->  
{lists:reverse(Odds), lists:reverse(Evens)}.
```

Чему нам удалось научиться

Теперь мы умеем создавать модули Эрланга и писать на нем простые последовательные программы. А также, мы почти уже готовы к написанию на нем более сложных последовательных программ.

Однако, следующая глава данной книги посвящена краткому рассмотрению темы работы с ошибками в Эрланге. После нее мы снова вернемся к последовательному программированию и рассмотрим в нем все, пока оставленные нами, детали.

Глава 4. Исключения

4.1. Исключения

Сообщения Эрланга об ошибках и реакции на них. До того как мы глубже погрузимся в последовательное программирование, давайте кратко рассмотрим эту тему. Это может показаться неожиданным отклонением от темы, но, если наша цель - писать надежные и распределенные выражения, то понимание того, как обрабатываются

ошибки становится просто необходимым.

Всякий раз когда мы вызываем функцию в Эрланге, происходит одно из двух: либо функция возвращает нам значение, или что-то идет не так. Мы видели такие примеры в предыдущей главе. Помните функцию cost?

```
cost(oranges) -> 5;
cost(newspaper) -> 8;
cost(apples) -> 2;
cost(pears) -> 9;
cost(milk) -> 7.
```

И вот что происходит при ее работе:

```
1> shop:cost(apples).
2
2> shop:cost(socks).
=ERROR REPORT==== 30-Oct-2006::20:45:10 ===
Error in process <0.34.0> with exit value:
{function_clause,[{shop,cost,[socks]},
{erl_eval,do_apply,5},
{shell,exprs,6},
{shell,eval_loop,3}]}
```

Когда мы вызвали cost(socks) функция обвалаилась (crashed). Это произошло потому что ни один из вариантов исполнения функции ("клозов") не подошел к данному аргументу функции.

Вызов cost(socks) -это полная бессмыслица. Функция не сможет вернуть нам никакого значения в ответ, поскольку цена на носки (socks) в ней просто не определена. В этом случае, вместо возврата значения, система запускает *исключение* - так, на техническом языке, называется "падение" программы.

Мы не пытаемся исправить эту ошибку, потому что это невозможно. Мы не знаем стоимость носков, а потому не можем вернуть никакое значение. Теперь это дело того кто вызвал такую функцию (cost(socks)) решать, что-же теперь делать дальше, когда функция "обвалилась".

Исключения запускаются системой, когда происходят внутренние ошибки или в самом коде, когда вызываются throw(Exception), exit(Exception). или erlang:error(Exception).

В Эрланге есть два метода *перехвата* исключений. Один из них - заключение вызова функции, которая может запустить исключение внутри **try...catch** выражения. Второй - это заключить такой вызов в **catch** выражение.

4.2. Запуск Исключений

Исключения запускаются автоматически, когда система обнаруживает какую-либо ошибку. Типичные ошибки - несоответствие образцу (включая отсутствие подходящих способов обработки аргументов функции), либо вызов стандартных BIF-функций с неправильным типом аргументов (например, вызов atom_to_list с целочисленным аргументом).

Кроме того, мы можем сами, непосредственно сгенерировать ошибку, вызвав одну из порождающих исключение BIF-функций:

```
exit(Why)
```

Она используется когда вы действительно хотите терминировать данный процесс.

Если соответствующее исключение не будет перехвачено, то сообщение вида {EXIT,Pid,Why} будет послано всем процессам, которые связаны с данным.

Подробнее мы поговорим об этом в разделе 9.1 Связанные процессы, поэтому здесь я не буду останавливаться на деталях.

```
throw(Why)
```

Эта функция используется для запуска исключения, которое вызывающий, возможно захочет перехватить. В этом случае мы должны указать в документации к нашей функции, что она может запускать исключение. У пользователя такой функции будет две альтернативы: либо программировать как обычно и просто слепо игнорировать это исключение, либо можно заключить вызов этой функции внутри try...catch выражения и обработать его.

```
erlang:error(Why)
```

Она используется чтобы обозначить "критическую" ошибку в программе. Так иногда бывает лучше, когда происходит что-то совсем нехорошее, с чем очень трудно справиться. Это эквивалентно сгенерированной внутренней ошибке.

Теперь давайте попробуем перехватить эти исключения.

4.3. try...catch

Если вы знакомы с языком Java, тогда выражение try...catch будет для вас весьма знакомо. Java перехватывает исключения с помощью следующего синтаксиса:

```
try {
    block
} catch (exception type identifier) {
    block
} catch (exception type identifier) {
    block
}
...
finally {
    block
}
```

В Эрланге эта конструкция исключительно похожа:

```
try FuncOrExpressionSequence of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
catch
    ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;
    ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;
    ...
after
    AfterExpressions
end
```

Заметьте схожесть между try...catch выражением и case выражением:

```
case Expression of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end
```

Выражение try...catch это как case выражение выращенное на стероидах. Оно очень похоже на case выражение, но с блоками catch и after в своем начале и конце.

У выражения try...catch есть значение

Вы не забыли? В Эрланге все является выражением и все выражения имеют значение. А значит и выражение try...catch тоже имеет свое значение. То есть мы можем написать что-то вроде:

```
f(...) ->
    ...
```

```
X = try
  ...
end,
Y = g(X),
...
```

Но, обычно, нам не надо значение выражения `try...catch`. Поэтому мы обычно пишем просто:

```
f(...) ->
  ...
  try
    ...
  end,
  ...
...
```

Выражение `try...catch` работает следующим образом: Первым делом вычисляется `FuncOrExpressionSeq`. Если его вычисление заканчивается без запуска исключения, тогда возвращенное оттуда значение проверяется на соответствие образцу `Pattern1` (с охранником `Guard1` если он присутствует), потом с образцом `Pattern2` и так далее, пока не будет найдено соответствие, и тогда, значением всего выражения `try...catch` будет значение вычисления выражения, следующего за подходящим образцом.

Если внутри `FuncOrExpressionSeq` будет запущено исключение, тогда на соответствие ему будут проявляться образцы `ExPattern1` и так далее, пока не будет найдено соответствие и ее последовательность выражений для вычисления. `ExceptionType` - это атом (один из `throw`, `exit` или `error`), который говорит нам, как это исключение было сгенерировано. Если `ExceptionType` отсутствует, то, по-умолчанию, считается, что он - `throw`.

Замечание: внутренние ошибки, обнаруженные системой исполнения приложений Эрланга, всегда имеют метку `error`.

Код, следующий за ключевым словом `after`, используется, обычно, для уборки после выполнения `FuncOrExpressionSeq`. Этот код гарантированно будет выполнен, даже если будет запущено исключение. Код в секции `after` будет запущен сразу после исполнения кода в секциях `try` или `catch`. Возвращаемое значение `AfterExpressions` будет утеряно.

Если вы пришли сюда со знанием языка Руби, то все это должно быть также весьма знакомо для вас - в Руби мы используем следующий тип выражения:

```
begin
```

```
    ...
rescue
    ...
ensure
    ...
end
```

И хотя ключевые слова отличаются, но общее поведение - очень похоже. (Хотя в Эрланге нет выражения `retry`!)

Сокращения

Некоторые части выражения `try...catch` могут быть опущены. Следующая запись

```
try F
catch
    ...
end
```

означает тоже самое что и:

```
try F of
    Val -> Val
catch
    ...
end
```

Аналогично, и раздел `after` может быть пропущен.

Программирование Идиом с `try...catch`

Когда мы разрабатываем приложение, мы часто хотим, чтобы код, перехватывающий ошибки, мог перехватить все ошибки, которые функция может сгенерировать.

Вот пара функций для иллюстрации этого. Первая функция генерирует все возможные типы исключений:

/файл `try_test.erl`/

```
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> erlang:error(a).
```

А теперь мы напишем вызывающую ее функцию внутри выражения try...catch.

/файл try_test.erl/

```
demo1() ->
    [catcher(I) || I <- [1,2,3,4,5]].

catcher(N) ->
    try generate_exception(N) of
        Val -> {N, normal, Val}
    catch
        throw:X -> {N, caught, thrown, X};
        exit:X -> {N, caught, exited, X};
        error:X -> {N, caught, error, X}
    end.
```

Запустив ее мы увидим следующее:

```
1> try_test:demo1().
[{1,normal,a},
 {2,caught,thrown,a},
 {3,caught,exited,a},
 {4,normal,['EXIT',a]},
 {5,caught,error,a}]
```

Получается, что мы можем перехватить и обработать все формы исключений, которые может сгенерировать нам функция.

4.4. catch

Другим способом перехватить исключение является использование примитива catch. Когда вы так перехватываете исключение, оно конвертируется в тьюпл, который описывает случившуюся ошибку. Чтобы продемонстрировать это, мы можем вызвать generate_exception внутри catch выражения:

```
demo2() ->
    [{I, (catch generate_exception(I))} || I <- [1,2,3,4,5]].
```

Запустив эту функцию, мы получим следующее:

```
2> try_test:demo2().
[{1,a},
 {2,a},
```

```
{3,['EXIT',a]},  
{4,['EXIT',a]},  
{5,['EXIT',{a,[{try_test,generate_exception,1},  
{try_test,'-demo2/0-fun-0-',1},  
{lists,map,2},  
{lists,map,2},  
{erl_eval,do_apply,5},  
{shell,exprs,6},  
{shell,eval_loop,3}]}]}]
```

Если вы сравните это с результатом работы try...catch , то увидите, что мы утратили много информации для анализа причин возникшей проблемы.

4.5. Улучшение сообщений об ошибках

Одним из способов использования функции erlang:error является улучшение информативности сообщений об ошибках. Приведем пример. Если мы вызовем math:sqrt(X) с отрицательным аргументом, то мы увидим следующее:

```
1> math:sqrt(-1).  
exited: {badarith,[{math,sqrt,[-1]},  
{erl_eval,do_apply,5},  
{shell,exprs,6},  
{shell,eval_loop,3}]}]
```

Но мы можем написать обертку для этого, которая улучшит сообщение об ошибке:

```
sqrt(X) when X < 0 ->  
    erlang:error({squareRootNegativeArgument, X});  
sqrt(X) ->  
    math:sqrt(X).  
  
2> lib_misc:sqrt(-1).  
exited: {{squareRootNegativeArgument,-1},  
[{lib_misc,sqrt,1},  
{erl_eval,do_apply,5},  
{shell,exprs,6},  
{shell,eval_loop,3}]}]
```

4.6. Стиль программирования try...catch

Так как-же нам обрабатывать ошибке на практике. Это, естественно, зависит от ситуации....

Код, часто возвращающий error

Если ваша функция не является гарантированно проходным случаем, то вам, вероятно, следует возвращать что-то вроде {ok, Value} или {error, Reason}, но помните, что это заставит всех вызывающих вашу функцию что-то сделать с возвращаемым значением. У вас будут, при этом, две альтернативы: либо вот так:

```
...
case f(X) of
  {ok, Val} ->
    do_something_with(Val);
  {error, Why} ->
    %% ... do something with the error ...
end,
...
...
```

что обработает оба возвращаемых типа значений, или вот так:

```
...
{ok, Val} = f(X),
do_something_with(Val);
...
```

что запустит исключение если функция f(X) вернет {error, ...} .

Код, где ошибки возможны, но редки

В этом случае типично написание кода, который ожидает и обрабатывает ошибки как в нижеследующем примере:

```
try my_func(X)
catch
  throw:{thisError, X} -> ...
  throw:{someOtherError, X} -> ...
end
```

А код, который ловит ошибки, должен при этом иметь соответствующие ветки throw :

```
my_func(X) ->
case ... of
```

```
...
... ->
... throw({thisError, ...})
... ->
... throw({someOtherError, ...})
```

4.7. Перехват всех возможных исключений

Если мы хотим перехватить все возможные ошибки, мы можем использовать следующий тип выражения:

```
try Expr
catch
  _:_ ->
    ... Code to handle all exceptions ...
end
```

Если мы опустим тип исключения и напишем вот так:

```
try Expr
catch
  _ -> ... Code to handle all exceptions ...
end
```

то мы *НЕ* перехватим все ошибки, потому что в этом случае типом исключений будет только `throw`, который действует по-умолчанию.

4.8. Обработка ошибок в старом и новом стилях

Этот раздел предназначен только для Эрланг-ветеранов!

`try...catch` это относительно новая конструкция, которая была введена для исправления дефектов механизма `catch...throw`. Если вы эрланговец старой закалки, который не читал его последней документации (типа меня), тогда вы автоматически пишете код наподобие вот такого:

```
case (catch foo(...)) of
  {'EXIT', Why} ->
  ...
Val ->
  ...
```

```
end
```

Обычно, это тоже корректно работает, но почти всегда лучше написать следующее:

```
try foo(...) of
    Val -> ...
  catch
    exit:
        Why ->
        ...
  end
```

Так что, вместо написания case (catch ...) of ..., пишите try ... of

4.9. Трассировка стека вызовов

Когда исключение перехвачено, мы можем получить текущий стек вызовов с помощью функции erlang:get_stacktrace(). Рассмотрим пример:

```
demo3() ->
  try
    generate_exception(5)
  catch
    error:X ->
      {X, erlang:get_stacktrace()}
  end.

1> try_test:demo3().
{a,[{try_test,generate_exception,1},
 {try_test,demo3,0},
 {erl_eval,do_apply,5},
 {shell,exprs,6},
 {shell,eval_loop,3}]}{}
```

Получаемая трассировка стека содержит список функций из стека, которым данная функция должна вернуть значение, если получится. Он почти совпадает с последовательностью вызовов функций, который привел нас к данной функции, но все вызовы с хвостовой рекурсией будут отсутствовать в этой трассировке (См. раздел 8.9 *Немного о хвостовой рекурсии*).

Сточки зрения исправления ошибок в нашей программе (дебагинга) только первые несколько строчек представляют тут для нас интерес. Начало трассировки стека говорит нам о том, что система повалилась во время вычисления функции

`generate_exception` (из модуля `try_test`) с одним аргументом. `try_test:generate_exception/1` была вероятно вызвана `try_test:demo3()` (мы не можем быть в этом абсолютно уверены, потому что `try_test:demo3()` могла вызвать некоторую другую функцию, которая сделала вызов с хвостовой рекурсией `try_test:generate_exception/1`, и в этом случае трассировка стека не покажет нам записей о этой промежуточной функции).

Глава 5. Расширенное последовательное программирование

В настоящее время мы успешно продвигаемся к пониманию последовательного программирования. В главе 3, *Последовательное Программирование*, рассматривались основы написания функций. Эта глава охватывает следующее:

BIFs. Сокращение от *built-in functions* (встроенные функции). BIFы - функции, которые являются частью языка Эрланг. Они выглядят так, как если бы были написаны на Эрланге, но фактически они реализованы как примитивные операции в виртуальной машине Эрланга.

Binaries. Бинарные последовательности - это тип данных, который мы используем для сохранения неструктурированных областей памяти рациональным образом.

Битовый синтаксис - это синтаксис шаблонов сопоставления, используемый для упаковки и распаковки битовых полей из бинарных последовательностей.

Дополнительные темы - здесь речь идет о небольшом числе тем, необходимых для совершенствования нашего мастерства последовательного Эрланга.

После того, как вы освоите эту главу, вы будете знать почти все, что нужно знать о последовательном Эрланге, и вы будете готовы окунуться в тайну параллельного программирования.

5.1. BIFs. Встроенные функции

BIFы - это функции, которые встроены в Эрланг. Они обычно решают задачи, недоступные программе на Эрланге. Например, невозможно преобразовать список в кортеж или найти текущее время и дату. Для выполнения таких операций мы вызываем BIF.

Для примера, BIF `tuple_to_list/1` конвертирует кортеж в список, а `time/0` возвращает текущее время суток в часах, минутах и секундах:

```
1> tuple_to_list({12,cat,"hello"}).  
[12,cat,"hello"]  
2> time().  
{20,0,3}
```

Все BIFы ведут себя, как будто они принадлежат модулю `erlang`, хотя наиболее распространенные BIFы (такие как `tuple_to_list`) автоматически импортируются, то есть мы можем вызывать их, написав `tuple_to_list(...)` вместо `erlang:tuple_to_list(...)`.

Вы найдете полный список всех BIFов на странице руководства `erlang` вашего дистрибутива Эрланг, или по адресу <http://www.erlang.org/doc/man/erlang.html>.

5.2. Binaries. Бинарные последовательности

Структуры данных, называемые бинарными последовательностями (binary), используются для хранения большого количества неструктурированных данных. Бинарные последовательности хранят данные намного более компактным образом, чем списки или кортежи, а среда выполнения оптимизирована для эффективного ввода и вывода бинарных последовательностей.

Бинарные последовательности записываются и отображаются как последовательности целых чисел или строк, заключенные в двойные треугольные скобки. Для примера:

```
1> <<5,10,20>>.  
<<5,10,20>>  
2> <<"hello">>.  
<<"hello">>
```

Когда вы используете в бинарных последовательностях целые числа, каждое из них должно быть в диапазоне от 0 до 255. Бинарная последовательность `<<"cat">>` - это сокращение для `<<99,97,116>>`; то есть бинарная последовательность составляется из ASCII-кодов символов в этой строке.

Если содержимое бинарной последовательности является печатаемой строкой, то оболочка отображает эту бинарную последовательность как строку; иначе она будет отображена как последовательность целых чисел.

Мы можем создать бинарную последовательность и извлечь элементы бинарной последовательности, используя BIF, или мы можем использовать битовый синтаксис (см. раздел 5.3, Битовый синтаксис). В этом разделе я буду говорить только о BIFax.

```
@spec func(Arg1, ..., Argn) -> Val
```

Что означает эта `@spec`?

Это является примером обозначения типов Эрланга, конвенцией документации, которая используется Эрланг-сообществом для описания (среди прочих вещей) типов аргументов и возвращаемых значений функции. Она должна быть достаточно само-документируемой, но тем, кому требуется больше деталей, следует обратиться к Приложению А.

BIFы, манипулирующие бинарными последовательностями

Следующие BIFы манипулируют бинарными последовательностями:

```
@spec list_to_binary(IoList) -> binary()
```

`list_to_binary` возвращает бинарную последовательность, сконструированную из целых чисел и бинарных последовательностей в `IoList`. Здесь `IoList` - это список, в котором элементами являются целые числа из диапазона `0..255`, бинарные последовательности, или `IoList`ы:

```
1> Bin1 = <<1,2,3>>.  
<<1,2,3>>  
2> Bin2 = <<4,5>>.  
<<4,5>>  
3> Bin3 = <<6>>.  
<<6>>  
4> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).  
<<1,2,3,1,2,3,4,5,4,6>>  
@spec split_binary(Bin, Pos) -> {Bin1, Bin2}
```

Разделяет бинарную последовательность `Bin` на две части по позиции `Pos`:

```
1> split_binary(<<1,2,3,4,5,6,7,8,9,10>>, 3).  
{<<1,2,3>>,<<4,5,6,7,8,9,10>>}  
@spec term_to_binary(Term) -> Bin
```

Конвертирует любой терм Эрланга в бинарную последовательность.

Бинарная последовательность, произведенная с помощью `term_to_binary` сохраняется в так называемом внешнем формате терма. Термы, которые были конвертированы в бинарные последовательности с использованием `term_to_binary` могут быть сохранены в файлы, переданы в сообщениях по сети, и т.д., а первоначальные терм, из которого они были сделаны, может быть восстановлен

позже. Это чрезвычайно полезно для сохранения сложных структур данных в файлы или отправки сложный структур данных на удаленные машины.

```
@spec binary_to_term(Bin) -> Term
```

Эта функция обратна функции `term_to_binary`:

```
1> B = term_to_binary({binaries, "are", useful}).  
<<123,104,3,100,0,8,98,105,97,114,105,101,115,107,  
0,3,97,114,101,100,0,6,117,115,101,102,117,108>>  
2> binary_to_term(B).  
{binaries, "are", useful}  
@spec size(Bin) -> Int
```

Возвращает число байт в бинарной последовательности.

```
1> size(<<1,2,3,4,5>>).  
5
```

5.3. Битовый синтаксис

Упаковка и распаковка 16-битных цветов

Мы начнем с очень простого примера. Предположим, мы хотим представить 16-битный RGB цвет. Мы решили выделить 5 бит для красного канала, 6 бит для зеленого канала и 5 бит для синего канала.

(Мы используем на один бит больше для зеленого канала потому, что человеческий глаз более чувствителен к зеленому цвету.)

Мы можем создать 16-битную область памяти `Mem`, содержащую одиночный RGB триплет следующим образом:

```
1> Red = 2.  
2  
2> Green = 61.  
61  
3> Blue = 20.  
20  
4> Mem = <<Red:5, Green:6, Blue:5>>.  
<<23,180>>
```

Записью в строке 4 мы создаем 2-байтную бинарную последовательность,

содержащую 16-битную величину. Оболочка печатает ее как `<<23,180>>`.

Для упаковки памяти мы просто написали выражение `<<Red:5, Green:6, Blue:5>>`.

Для распаковки слова мы пишем шаблон:

```
5> <<R1:5, G1:6, B1:5>> = Mem.  
<<23,180>>  
6> R1.  
2  
7> G1.  
61  
8> B1.  
20
```

Выражения битового синтаксиса

Выражения битового синтаксиса имеют следующую форму:

```
<<>>  
<<E1, E2, ..., En>>
```

Каждый элемент `Ei` определяет единичный сегмент бинарной последовательности.

Каждый элемент `Ei` может иметь одну из четырех возможных форм:

```
Ei = Value |  
Value:Size |  
Value/TypeSpecifierList |  
Value:Size/TypeSpecifierList
```

Какую бы форму вы не использовали, общее число битов в бинарной последовательности должно быть кратно 8. (Это происходит потому, что бинарные последовательности содержат байты, которые имеют по 8 битов каждый, поэтому нет способа представления последовательности битов, чья длина не делится на 8.)

Когда вы конструируете бинарную последовательность, `Value` должна быть связанной переменной, символьной строкой или выражением, которое вычисляется в целое число, вещественное число или в бинарную последовательность. Когда используется операция сопоставления шаблона, `Value` может быть связанной или несвязанной переменной, целым, символьной строкой, действительным числом или бинарной последовательностью.

`Size` должен быть выражением, которое вычисляется в целое число. В шаблонах сопоставления `Size` должен быть целым числом или связанной переменной, величина

которой есть целое число. `Size` не может быть несвязанной переменной.

Величина `Size` определяет размер сегмента в блоках (мы обсудим это позже). Величина по умолчанию зависит от типа (смотри ниже). Для целых чисел это 8, для действительных чисел это 64, а для бинарной последовательности это размер бинарной последовательности. В шаблонах сопоставления эта величина по умолчанию корректна только для самого последнего элемента. Все другие элементы бинарной последовательности должны иметь спецификацию размера.

`TypeSpecifierList` - это дефисно-разделенный список элементов формы `End-Sign-Type-Unit`. Любой из предыдущих элементов может быть опущен, а элементы могут следовать в любом порядке. Если элемент опущен, для этого элемента используется величина по умолчанию.

Элементы в списке спецификаторов могут иметь следующие величины:

```
@type End = big | little | native
```

(`@type` - это также часть определения типов Эрланга, приведенного в Приложении А).

Этот элемент определяет порядок следования байтов машины. `native` - определяется во время выполнения, в зависимости от процессора вашей машины. Величина по умолчанию - `big`. Это имеет отношение только к упаковке и распаковке целых чисел из бинарных последовательностей. Когда упаковываются и распаковываются целые числа из бинарных последовательностей на машинах с различным порядком следования байтов, вы должны позаботиться о корректном порядке следования байтов.

Подсказка: В редком случае, если вам действительно нужно понять, что происходит, могут понадобиться некоторые эксперименты. Чтобы гарантировать себе, что вы делаете все правильно, попробуйте следующую команду оболочки:

```
1>{<<16#12345678:32/big>>, <<16#12345678:32/little>>,
   <<16#12345678:32/native>>, <<16#12345678:32>>}.
{<<18,52,86,120>>, <<120,86,52,18>>, <<120,86,52,18>>, <<18,52,86,120>>}
```

Этот вывод показывает вам, как именно целые числа упаковываются в бинарную последовательность с использованием битового синтаксиса.

Если вы все еще беспокоитесь, что `term_to_binary` и `binary_to_term` "делают правильные вещи" при упаковке и распаковке целых чисел, дополнительно проверьте их. Вы можете, для примера, создать кортеж, содержащий целые числа на машине с порядком следования байтов "от старшего к младшему" (big-endian). Затем используйте `term_to_binary` для конвертации терма в бинарную последовательность

и отправки ее в машину с порядком следования "от младшего к старшему" (little-endian). На машине с little-endian, выполните `binary_to_term`, и все целые числа в кортеже будут иметь правильные значения.

```
@type Sign = signed | unsigned
```

Знаковое | Беззнаковое.

Этот параметр используется только в шаблонах сопоставления. Значение по умолчанию - `unsigned` (беззнаковое).

```
@type Type = integer | float | binary
```

Тип. Целое | Вещественное | Бинарная последовательность

Значение по умолчанию - `integer`.

```
@type Unit = 1 | 2 | ... 255
```

Размер блока. Общий размер сегмента равен `Size` x `Unit` битов. Общий размер сегмента должен быть больше или равен нулю и должен быть кратным 8.

Значение по умолчанию для размера блока `Unit` зависит от типа `Type` и равно 1 если тип - целое или вещественное число (`integer` или `float`), либо 8, если тип - бинарная последовательность (`binary`).

Если вы нашли описание битового синтаксиса несколько устрашающим, не паникуйте. Получение правильных шаблонов битового синтаксиса довольно сложно. Лучший способ подхода к этому заключается в экспериментах в оболочке с шаблонами, которые вам необходимы, пока вы не получите их правильными, а затем вырежьте и вставьте результат в вашу программу. Вот как я это делаю.

Расширенные примеры битового синтаксиса

Изучение битового синтаксиса сложно, но преимущества огромны. Этот раздел содержит три примера из реальной жизни. Весь код здесь вырезан и вставлен из программ реального мира. Этими примерами являются:

- Нахождение кадра синхронизации в MPEG данных;
- Распаковка COFF данных;
- Распаковка заголовка в IPv4 дейтаграмме.

Нахождение кадра синхронизации в MPEG данных

Предположим, мы хотим написать программу, которая манипулирует аудиоданными MPEG. Мы могли бы написать потоковый медиа-сервер на Эрланге или извлечь тэги данных, которые описывают содержимое аудиопотока MPEG. Чтобы сделать это, нам необходимо идентифицировать и синхронизироваться с фреймами данных в MPEG потоке.

Аудиоданные MPEG складываются из ряда фреймов. Каждый фрейм имеет свой собственный заголовок, следующий перед аудио-информацией - имеется в виду не заголовок файла, и в принципе вы можете разрезать MPEG файл на части и воспроизвести любую из этих частей. Любая программа, которая читает MPEG поток должна найти заголовки фреймов и впоследствии синхронизировать MPEG данные.

Заголовок MPEG начинается с 11-битной кадровой синхронизации, состоящей из одиннадцати последовательных единиц, следующих перед информацией, которая описывает данные:

AAAAAAA AAABCCD EEEFFGH IIJKLMM

AAAAAAA Слово синхронизации (11 бит, все единицы)

BB 2 бита - идентификатор версии MPEG

CC 2 бита - описание уровня

D 1 бит - защитный бит

Точные детали этих битов нас здесь не касаются. В основном, с учетом знания значений от A до M мы можем вычислить длину MPEG фрейма.

Для нахождения точки синхронизации сначала предполагаем, что мы правильно позиционированы в начало MPEG фрейма. Мы используем информацию, которую нашли в этой позиции для вычисления длины фрейма. Мы могли бы указывать на бессмыслицу в случае, если длина фрейма будет полностью неправильной. Предполагая, что мы находимся в начале фрейма и задана длина фрейма, можем перескочить в начало следующего фрейма и увидеть, является ли это другим заголовком MPEG фрейма.

Для нахождения точки синхронизации сначала предполагаем, что мы правильно позиционированы в начало заголовка MPEG. Затем попытаемся вычислить длину фрейма. Может произойти один из следующих случаев:

Наше предположение было правильным, поэтому, когда мы перескочим вперед на длину фрейма, мы найдем другой заголовок MPEG.

Наше предположение было некорректным; либо мы не позиционированы на

последовательность из 11 единиц, которые помечают начало фрейма, либо формат слова некорректный, так что мы не можем вычислить длину фрейма.

Наше предположение было некорректным, но мы позиционированы на паре байтов музыкальных данных, которые случайно выглядят как заголовок фрейма. В этом случае мы можем вычислить длину фрейма, но не сможем найти новый заголовок.

Чтобы быть действительно уверенными, мы просмотрим три последовательных заголовка. Функция синхронизации:

Скачать [mp3_sync.erl](#)

```
find_sync(Bin, N) ->
    case is_header(N, Bin) of
        {ok, Len1, _} ->
            case is_header(N + Len1, Bin) of
                {ok, Len2, _} ->
                    case is_header(N + Len1 + Len2, Bin) of
                        {ok, _, _} ->
                            {ok, N};
                        error ->
                            find_sync(Bin, N+1)
                    end;
                error ->
                    find_sync(Bin, N+1)
            end;
        error ->
            find_sync(Bin, N+1)
    end;
    error ->
        find_sync(Bin, N+1)
end.
```

`find_sync` пытается найти три последовательных заголовка MPEG фреймов. Если байт `N` в `Bin` является началом заголовка фрейма, то `is_header(N, Bin)` вернет `{ok, Length, Info}`. Если `is_header` возвращает `error`, то `N` не может указывать на начало правильного фрейма. Мы можем сделать быстрый тест в оболочке, чтобы убедиться, что это работает:

```
1> {ok, Bin} = file:read_file("/home/joe/music/mymusic.mp3").
{ok,<<73,68,51,3,0,0,0,0,33,22,84,73,84,50,0,0,0,28, ...>}
2> mp3_sync:find_sync(Bin, 1).
{ok,4256}
```

При этом используется `file:read_file` для чтения всего файла в бинарную последовательность (смотри раздел 13.2, Чтение всего файла в бинарную последовательность). Код для `is_header`:

[Скачать mp3_sync.erl](#)

```
is_header(N, Bin) ->
    unpack_header(get_word(N, Bin)).

get_word(N, Bin) ->
    {_, <<C:4/binary, _/binary>>} = split_binary(Bin, N),
    C.

unpack_header(X) ->
    try decode_header(X)
    catch
        _:_ -> error
    end.
```

Это немного сложнее. Сначала мы извлекаем 32 бита данных для анализа (это делается с помощью `get_word`); затем мы распаковываем заголовок с использованием `decode_header`. `decode_header` написан так, чтобы рушиться (вызовом `exit/1`) если его аргумент не является началом заголовка. Чтобы перехватить все ошибки мы оборачиваем вызов `decode_header` в конструкцию `try...catch` (прочтите об этом больше в разделе 4.1, *Исключения*). Этим так же будут перехвачены все ошибки, которые могли произойти из-за некорректного кода в функции `framelen/4`. Код функции `decode_header`, в которой начинается все веселье:

[Скачать mp3_sync.erl](#)

```
decode_header(<<2#111111111111:11, B:2, C:2, _D:1, E:4, F:2, G:1, Bits:9>>) ->
    Vsn = case B of
        0 -> {2,5};
        1 -> exit(badVsn);
        2 -> 2;
        3 -> 1
    end,
    Layer = case C of
        0 -> exit(badLayer);
        1 -> 3;
        2 -> 2;
        3 -> 1
    end,
    %% Protection = D,
    BitRate = bitrate(Vsn, Layer, E) * 1000,
    SampleRate = samplerate(Vsn, F),
    Padding = G,
```

```

FrameLength = framelength(Layer, BitRate, SampleRate, Padding),
if
    FrameLength < 21 ->
        exit(frameSize);
    true ->
        {ok, FrameLength, {Layer, BitRate, SampleRate, Vsn, Bits}}
end;
decode_header(_) ->
    exit(badHeader).

```

Этот шаблон сопоставляет 11 последовательных единичных битов¹, 2 бита в B, 2 бита в C и так далее. Обратите внимание, что код в точности соответствует спецификации битового уровня MPEG заголовка, данной ранее. Более красивый и прямой код написать будет трудно. Этот код прекрасен. Также он очень эффективен. Компилятор Эрланга переводит шаблоны битового синтаксиса в высоко оптимизированный код, который извлекает значения полей оптимальным образом.

Распаковка COFF данных

Несколько лет назад я решил написать программу для создания автономных Эрланг-программ, которые будут работать в Windows - я хотел собирать исполняемые модули Windows на любой машине, которая могла запустить Эрланг. Выполнение этого повлекло за собой понимание и манипулирование файлами, имеющими формат Microsoft Common Object Format (COFF). Выяснить подробности COFF было довольно сложно, но различные API для C++ программ были задокументированы. C++ программы использовали объявления типов DWORD, LONG, WORD и BYTE (эти объявления типов будут знакомы программистам, которые программировали внутренности Windows).

Структуры данных были задокументированы, но только с точки зрения программистов С и C++. Ниже приводится типичный С `typedef`:

```

typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    WORD NumberOfNamedEntries;
    WORD NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;

```

Для написания моей Эрланг программы я, прежде всего, определил четыре макроса, которые должны быть включены в файл с исходным кодом на Эрланге:

```
-define(DWORD, 32/unsigned-little-integer).
#define(LONG, 32/unsigned-little-integer).
#define(WORD, 16/unsigned-little-integer).
#define(BYTE, 8/unsigned-little-integer).
```

Примечание: макросы описаны в разделе 5.4, *Макросы*. Для раскрытия этих макросов мы используем синтаксис ?DWORD, ?LONG и так далее. К примеру, макрос ?DWORD разворачивается в символьный текст `32/unsigned-little-integer`.

Эти макросы преднамеренно имеют такие же имена, как и их коллеги на С. Вооруженный этими макросами, я мог бы легко написать код для распаковки данных графических ресурсов в бинарную последовательность:

```
unpack_image_resource_directory(Dir) ->
    <<Characteristics : ?DWORD,
    TimeDateStamp : ?DWORD,
    MajorVersion : ?WORD,
    MinorVersion : ?WORD,
    NumberOfNamedEntries : ?WORD,
    NumberOfIdEntries : ?WORD, _/binary>> = Dir,
    ...
```

Если вы сравните код на С и на Эрланге, то вы увидите, что они очень похожи. Итак, позаботясь об именах макросов и разбиение кода на Эрланге, мы можем свести к минимуму семантический разрыв между кодом на С и кодом на Эрланге, что делает нашу программу более легкой к пониманию и меньше подверженной ошибкам.

Следующим шагом стала распаковка данных по характеристикам и т.д.

Характеристики - это 32-битное слово, состоящее из набора флагов. Их распаковка, с использованием битового синтаксиса, выполняется очень просто; мы просто должны написать код, подобный следующему:

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =
    <<Characteristics:32>>
```

Код `<<Characteristics:32>>` конвертирует характеристики, которые были целым числом, в бинарную последовательность длиной 32 бита. Затем следующий код распаковывает требуемые биты в переменные `ImageFileRelocsStripped`, `ImageFileExecutableImage` и так далее:

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =
```

Опять же, я оставил все имена такими же, как в Windows API, чтобы свести

семантический разрыв между спецификацией и программой на Эрланге к минимуму.

С помощью этих макросов сделал распаковку данных в COFF формат - хорошо, я не могу использовать слово проще - но по крайней мере это было возможно и код был вполне понимаемым.

Распаковка заголовка в IPv4 дейтаграмме

Этот пример иллюстрирует разбор дейтаграмм протокола Internet Protocol версии 4 (IPv4) в одной операции сопоставления шаблона:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

...
DgramSize = size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
     ID:16, Flgs:3, FragOff:13,
     TTL:8, Proto:8, HdrChkSum:16,
     SrcIP:32,
     DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<0pts:OptsLen/binary,Data/binary>> = RestDgram,
    ...

```

Этот код сопоставляет IP дейтаграмму в одном выражении шаблонного сопоставления. Шаблон является сложным, продолжающимся на трех строках, и иллюстрирует каким образом данные, не попадающие на границы байтов, легко могут быть извлечены (например, поля `Flgs` и `FragOff` длиной 3 и 13 бит соответственно). Имея шаблон сопоставления IP дейтаграммы, заголовок и часть данных дейтаграммы извлекаются во второй операции шаблонного сопоставления.

5.4. Дополнительные короткие темы

Сейчас мы охватили все основные темы последовательного Эрланга. То, что осталось, это несколько небольших странностей и дополнений, которые вы должны знать, но которые не подходят ни под одну из других тем. В них нет особенного логического порядка. Эти темы охватываются следующим образом:

- *apply*: Как вычисляется значение функции по ее имени и аргументам, когда имена функции и модуля вычисляются динамически.
- *Атрибуты*: Синтаксис и смысл атрибутов модуля на Эрланге.

- *Блоки выражений*: Выражения, использующие `begin` и `end`.
- *Булевые выражения*: Все булевые выражения.
- *Набор символов*: Какой набор символов используется в Эрланге.
- *Комментарии*: Синтаксис комментариев.
- `epp`: Препроцессор Эрланга.
- *Escape-последовательности*: Синтаксис управляющих последовательностей, используемых в строках и атомах.
- *Выражения и последовательности выражений*: Что именно представляет из себя выражение?
- *Функциональные ссылки*: Как ссылаться на функции.
- *Включаемые файлы*: Как включать файлы во время компиляции.
- *Операции со списками*: `++` и `--`.
- *Макросы*: Макро-процессор Эрланга.
- *Оператор сопоставления в шаблонах*: Как оператор сопоставления `=` может быть использован в шаблонах.
- *Числа*: Синтаксис чисел.
- *Старшинство операторов*: приоритет и ассоциативность всех операторов Эрланга.
- *Словарь процесса*: Каждый процесс Эрланга имеет локальную область разрушающей памяти, которая иногда может быть полезной.
- *Ссылки*: Ссылки - это уникальные символы.
- *Упрощенные булевые выражения*: Булевые выражения, которые не вычисляются полностью.
- *Сравнения термов*: Все операторы сравнения термов и лексического упорядочения термов.
- *Подчеркнутые переменные*: Переменные, которые компилятор рассматривает особым образом.

apply

BIF `apply(Mod, Func, [Arg1, Arg2, ..., ArgN])` применяет функцию `Func` из модуля `Mod` с аргументами `Arg1, Arg2, ..., ArgN`. Это эквивалентно вызову:

```
Mod:Func(Arg1, Arg2, ..., ArgN)
```

`apply` позволяет вам вызывать функцию в модуле, передавая ей аргументы. Что отличает ее от прямого вызова функции, это то, что имя модуля и/или функции может быть вычислено динамически.

Все BIFы Эрланга могут быть вызваны с использованием `apply` с предположением,

что они принадлежат модулю `erlang`. Так, для создания динамического вызова BIF, мы можем написать следующее:

```
1> apply(erlang, atom_to_list, [hello]).  
"hello"
```

Предупреждение: Если это возможно, следует избегать использования `apply`. Когда число аргументов функции известно заранее, намного лучше использовать вызов формы: `M:F(Arg1, Arg2, ..., ArgN)`, чем `apply`. Когда вызовы функций строятся с использованием `apply`, многие инструменты анализа не могут понять, что происходит, а некоторые компиляторные оптимизации не могут быть выполнены. Итак, используйте `apply` редко и только когда это абсолютно необходимо.

Атрибуты

Атрибуты модуля имеют синтаксис `-AtomTag(...)`² и используются для определения некоторых свойств файла. Существует два типа атрибутов модуля: предопределенные и пользовательские.

Предопределенные атрибуты модуля

Следующие атрибуты модуля имеют предопределенные значения и должны быть помещены перед любым определением функции.

`-module(modname).` Декларация модуля. modname должен быть атомом. Этот атрибут должен быть первым атрибутом в файле. Общепринято, что код для modname должен быть сохранен в файле с именем `modname.erl`. Если вы не сделаете этого, автоматическая загрузка кода не будет работать корректно; подробнее смотри в разделе E.4, *Динамическая загрузка кода*.

`-import(Mod, [Name1/Arity1, Name2/Arity2, ...]).` Определяет, что функция Name1 с числом аргументов Arity1 является импортированной из модуля Mod.

Если функция однажды импортирована из модуля, то вызов функции может быть выполнен без указания имени модуля.

Например:

```
-module(abc).  
-import(lists, [map/2]).  
  
f(L) ->  
L1 = map(fun(X) -> 2*X end, L),  
lists:sum(L1).
```

Вызов тар не требует указания имени модуля, в то время как для вызова sum нам необходимо включить имя модуля в вызове функции.

`-export([Name1/Arity1, Name2/Arity2, ...]).` Экспортирует функции Name1/Arity1, Name2/Arity2, и т.д. из текущего модуля. Заметьте, что только экспортированные функции могут быть вызваны снаружи модуля. Для примера:

[Скачать abc.erl](#)

```
-module(abc).  
-export([a/2, b/1]).  
a(X, Y) -> c(X) + a(Y).  
a(X) -> 2 * X.  
b(X) -> X * X.  
c(X) -> 3 * X.
```

Декларация экспорта указывает, что только `a/2` и `b/1` могут быть вызваны вне модуля `abc`. Так например, вызов `abc:a(5)` приведет к ошибке, т.к. `a/1` не экспортирована из модуля.

```
1> abc:a(1,2).  
7  
2> abc:b(12).  
144  
3> abc:a(5).  
exited: {undef,[{abc,a,[5]},  
{erl_eval,do_apply,5},  
{shell,exprs,6},  
{shell,eval_loop,3}]} ="session">
```

`-compile(Options).` Добавляет `Options` к списку опций компилятора. `Options` - это одиночная опция компилятора или список опций компилятора (они описаны на странице руководства для модуля `compile`).

Примечание: Опция компилятора `-compile(export_all)`. наиболее часто используется при отладке программ. Она экспортирует все функции из модуля без подробного использования аннотации `-export`.

`-vsn(Version).` Определяет версию модуля. `Version` - любой символьный терм. Значение `Version` не имеет специального синтаксиса или смысла, но оно может быть использовано анализирующими программами или для целей документирования.

Пользовательские атрибуты

Синтаксис пользовательских атрибутов модуля следующий:

```
-SomeTag(Value).
```

`SomeTag` должен быть атомом, а `Value` должна быть символьным термом. Значения атрибутов модуля компилируются в модуль и могут быть извлечены во время выполнения. Вот пример:

```
-module(attrs).
-vsn(1234).
-author({joe,armstrong}).
-purpose("example of attributes").
-export([fac/1]).

fac(1) -> 1;
fac(N) -> N * fac(N-1).

1> attrs:module_info().
[{exports,[{fac,1},{module_info,0},{module_info,1}]},
 {imports,[]},
 {attributes,[{vsn,[1234]},
   {author,[{joe,armstrong}]},
   {purpose,"example of attributes"}]},
 {compile,[{options,[{cwd,"/home/joe/2006/book/JAERLANG/Book/code"},{outdir,"/home/joe/2006/book/JAERLANG/Book/code"}]}, {version,"4.4.3"}, {time,{2007,2,21,19,23,48}}, {source,"/home/joe/2006/book/JAERLANG/Book/code/attrs.erl"}]}
2> attrs:module_info(attributes).
[{vsn,[1234]},{author,[{joe,armstrong}]},{purpose,"example of attributes"}]
3> beam_lib:chunks("attrs.beam",[attributes]).
{ok,{attrs,[{attributes,[{author,[{joe,armstrong}]}],{purpose,"example of attributes"},{vsn,[1234]}]}]}}
```

Пользовательские атрибуты, содержащиеся в файле исходного кода, повторяются как подтермы `{attributes, ...}`. Кортеж `{compile, ...}` содержит информацию, которая была добавлена компилятором. Значение `{value,"4.4.3"}` является версией компилятора и ее не следует путать с тегом `vsn`, определенным в атрибутах модуля. В предыдущем примере `attrs:module_info()` возвращает список свойств всех метаданных, ассоциированных с откомпилированным модулем, `attrs:module_info(attributes)`³ возвращает список всех атрибутов, ассоциированных с файлом.

Обратите внимание, что функции `module_info/0` и `module_info/1` автоматически создаются каждый раз при компиляции модуля.

Вывод в строках 2 и 3 немного тяжело читать. Чтобы сделать жизнь проще, вы можете написать небольшую функцию, которая извлекает конкретный атрибут и вызвать ее, как показано ниже:

```
4> extract:attribute("attrs.beam", author).
[{:joe, armstrong}]
```

Код, выполняющий это прост:

[Скачать extract.erl](#)

```
-module(extract).
-export([attribute/2]).
attribute(File, Key) ->
    case beam_lib:chunks(File, [attributes]) of
        {ok, {_Module, [{attributes, L}]}} ->
            case lookup(Key, L) of
                {ok, Val} ->
                    Val;
                error ->
                    exit(badAttribute)
            end;
        _ ->
            exit(badFile)
    end.

lookup(Key, [{Key, Val}|_]) -> {ok, Val};
lookup(Key, [_|T]) -> lookup(Key, T);
lookup(_, []) -> error.
```

Чтобы запустить `attrs:module_info`, мы должны загрузить байт-код для модуля `attrs`. Модуль `beam_lib` содержит ряд функций для анализа модуля без загрузки кода. Например, в `extract.erl` используется `beam_lib:chunks` для извлечения данных атрибута без загрузки кода модуля.

Блок выражений

```
begin
    Expr1,
    ...,
    ExprN
end
```

Вы можете использовать блок выражений для группировки последовательности выражений, аналогично телу условия. Значением блока `begin ... end` является значение последнего выражения в блоке.

Блок выражений используется когда синтаксис требует одиночное выражение, но вы хотите иметь последовательность выражений в этом месте кода.

Булевые значения

В Эрланге не существует специального булева типа; вместо этого особую интерпретацию получают атомы `true` и `false` и они используются для представления булевых символов.

Булевые выражения

Существует четыре булевых выражения:

- `not B1`: Логическое НЕ;
- `B1 and B2`: Логическое И;
- `B1 or B2`: Логическое ИЛИ;
- `B1 xor B2`: Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ.

Во всех них `B1` и `B2` должны быть булевыми символами или выражениями, которые имеют булевые значения. Примеры:

```
1> not true.  
false.  
2> true and false.  
false  
3> true or false.  
true  
4> (2 > 1) or (3 > 4).  
true
```

Усиление бинарных функций возвращением булевых значений

Иногда мы пишем функции, которые возвращают одно из двух возможных атомарных значений. Когда это происходит, правильной практикой будет убедиться, чтобы они возвращают булевые значения. Кроме того, хорошая идея именовать ваши функции так, чтобы можно было понять, что они возвращают логическое значение.

Например, предположим мы пишем программу, которая представляет состояние некоторого файла. Мы могли бы обнаружить себя пишущими функцию `file_state()`,

которая возвращает `open` и `closed`. Когда мы пишем эту функцию, мы могли бы подумать о переименовании этой функции и позволить ей возвращать булево значение. Немного поразмыслив, мы перепишем нашу программу для использования функции `is_file_open`, которая возвращает `true` или `false`.

Почему мы должны сделать это?

Ответ прост. Есть большое количество функций в стандартных библиотеках, которые работают на функциях, возвращающих булевые значения. Поэтому, если мы убедимся, что все наши функции, которые могут возвращать одно из двух атомарных значений вместо этого возвращают булевые значения, то мы будем способны использовать их совместно со стандартными библиотечными функциями.

Набор символов

Предполагается, что файлы с исходными текстами Эрланга кодируются в наборе символов **ISO-8859-1 (Latin-1)**. Это означает, что все печатаемые символы **Latin-1** могут быть использованы без использования любыми управляемыми последовательностями.

Внутри Эрланг не имеет символьного типа данных. Строки реально не существуют, вместо этого они представляются списками целых чисел. Строки Юникод (Unicode) могут быть представлены списками целых чисел без каких-либо проблем, хотя существует ограниченная поддержка разбора и создания файлов в кодировке Юникод из целочисленных списков Эрланга.

Комментарии

Комментарии в Эрланге начинаются с символа процента `%` и продолжаются до конца строки. Блочных комментариев нет.

Примечание: Вы часто будете видеть двойные символы процента `%%` в примерах кода. Двойные знаки процента распознаются в erlang-mode Emacs и разрешают автоматический отступ закомментированных строк.

```
% This is a comment
my_function(Arg1, Arg2) ->
    case f(Arg1) of
        {yes, X} -> % it worked
        ...
    end.
```

epp

Перед тем, как модуль Эрланга будет откомпилирован, он автоматически

обрабатывается препроцессором Эрланга epp. Препроцессор раскрывает все макросы, которые могут быть в исходном файле и вставляет все необходимые включаемые файлы.

Обычно вам не нужно наблюдать вывод препроцессора, но в исключительных обстоятельствах (например, при отладке неисправного макроса) вы можете захотеть сохранить вывод препроцессора. Вывод препроцессора может быть сохранен в файл с помощью команды `compile:file(M, ['P'])`. Этим компилируется любой код в файле M.erl и выполняется листинг в файл M.P, где все макросы были раскрыты и все необходимые включаемые файлы были вставлены.

Управляющие последовательности.

В строках и атомах с кавычками вы можете использовать управляющие последовательности для ввода каких-либо непечатаемых символов. Все возможные управляющие последовательности показаны в таблице 5.1.

Приведем несколько примеров в оболочке, чтобы показать, как эти конвенции работают. (Примечание: `~w` в строке формата печатает список без каких-либо попыток напечатать достаточный результат.)

```
% Control characters
1> io:format("~w~n", ["\b\d\e\f\n\r\s\t\v"]).
[8,127,27,12,10,13,32,9,11]
ok
% Octal characters in a string
3> io:format("~w~n", ["\123\12\1"]).
[83,10,1]
ok
% Quotes and escapes in a string
4> io:format("~w~n", ["'\"\\\""]).
[39,34,92]
ok
% Character codes
5> io:format("~w~n", ["\a\z\A\Z"]).
[97,122,65,90]
ok
```

Управляющая последовательность	Значение	Числовой код
<code>\b</code>	Возврат на одну позицию (Backspace)	8
<code>\d</code>	Удалить	127

<code>\e</code>	Выход (Escape)	27
<code>\f</code>	Прогон страницы (Form feed)	12
<code>\n</code>	Перевод строки	10
<code>\r</code>	Возврат каретки	13
<code>\s</code>	Пробел	32
<code>\t</code>	Табуляция (Tab)	9
<code>\v</code>	Вертикальная табуляция	11
<code>\NNN</code> <code>\NN</code> <code>\N</code>	Восьмиричные символы (N есть 0..7)	
<code>^a..^z</code> или <code>^A..^Z</code>	От Ctrl+A до Ctrl+Z	1..26
<code>\'</code>	Одиночный апостроф	39
<code>\"</code>	Двойной апостроф	34
<code>\\"</code>	Обратный слэш	92
<code>\C</code>	Код ASCII для С (С - символ)	Целое число

Таблица 5.1. Управляющие последовательности.

Выражения и последовательности выражений

В Эрланге все, что может быть вычислено для производства значения называется *выражение*. Это означает, что такие вещи, как `catch`, `if` и `try...catch` являются выражениями. Такие вещи, как записи или атрибуты модулей не могут быть вычислены, поэтому они не являются выражениями.

Последовательности выражений - это ряды выражений, разделенных запятыми. Они находятся повсюду сразу после стрелки `->`. Значение последовательности выражений `E1, E2, ..., En` определяется значением последнего выражения в последовательности⁴. Это значение вычисляется с использованием любых привязок, созданных при вычислении значений `E1`, `E2` и т.д.

Функциональные ссылки

Часто мы хотим сослаться на функцию, которая определена в текущем модуле или в каком-либо внешнем модуле. Вы можете использовать следующее обозначение для этого:

```
fun LocalFunc/Arity
```

Используется для ссылки на локальную функцию с именем `LocalFunc` и числом

аргументов `Arity` в текущем модуле.

```
fun Mod:RemoteFunc/Arity
```

Используется для ссылки на внешнюю функцию с именем `RemoteFunc` и числом аргументов `Arity` в модуле `Mod`.

Пример функциональной ссылки в текущем модуле:

```
-module(x1).  
-export([square/1, ...]).  
  
square(X) -> X * X.  
...  
double(L) -> lists:map(fun square/1, L).
```

Если мы хотим вызвать функцию в удаленном модуле, мы можем сослаться на функцию, как в следующем примере:

```
-module(x2).  
  
...  
double(L) -> lists:map(fun x1:square/1, L).
```

`fun x1:square/1` означает функцию `square/1` в модуле `x1`.

Включаемые файлы

Файлы могут быть включены с использованием следующего синтаксиса:

```
-include(Filename).
```

В Эрланге есть договоренность, что включаемые файлы имеют расширение `.hrl`. `FileName` должно содержать абсолютный или относительный путь, по которому препроцессор может обнаружить соответствующий файл. Файлы библиотечных хедеров могут быть включены с использованием следующего синтаксиса:

```
-include_lib(Name).
```

Например:

```
-include_lib("kernel/include/file.hrl").
```

В этом случае компилятор Эрланга будет искать соответствующие включаемые

файлы. (`kernel` в предыдущем примере ссылается на приложение, в котором определен этот заголовочный файл.)

Включаемые файлы обычно содержат определения записей. Если нескольким модулям необходимо разделять общие определения записей, то эти общие определения записей помещаются во включаемые файлы, которые включаются всеми модулями, которым необходимы данные определения.

Операции со списками ++ и --

`++` и `--` являются инфиксными операторами для сложения и вычитания списков.

`A ++ B` складывает (т.е. присоединяет) A и B.

`A -- B` вычитает список B из списка A. Вычитание означает, что каждый элемент B удаляется из A. Обратите внимание, что если некоторый символ X входит K раз в B, то только первые K вхождений X в A будут удалены.

Примеры:

```
1> [1,2,3] ++ [4,5,6].  
[1,2,3,4,5,6]  
2> [a,b,c,1,d,e,1,x,y,1] -- [1].  
[a,b,c,d,e,1,x,y,1]  
3> [a,b,c,1,d,e,1,x,y,1] -- [1,1].  
[a,b,c,d,e,x,y,1]  
4> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1].  
[a,b,c,d,e,x,y]  
5> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1].  
[a,b,c,d,e,x,y]
```

++ в шаблонах

`++` может быть использован в шаблонах. Когда сопоставляются строки, мы можем писать такие шаблоны, как в следующем примере:

```
f("begin" ++ T) -> ...  
f("end" ++ T) -> ...  
...
```

Шаблон в первом случае расширяется в `[$b,$e,$g, $i,$n|T]`.

Макросы

Макросы в Эрланге записываются, как показано здесь:

```
-define(Constant, Replacement).  
-define(Func(Var1, Var2, ..., Var), Replacement).
```

Макросы раскрываются препроцессором Эрланга `erl`, когда встречается выражение формы `?MacroName`. Переменные, встречающиеся в определении макроса, полностью совпадают по форме в соответствующем месте вызова макроса.

```
-define(macro1(X, Y), {a, X, Y}).  
foo(A) ->  
    ?macro1(A+10, b)
```

Этот пример раскладывается в:

```
foo(A) ->  
    {a,A+10,b}.
```

в дополнение, существует несколько предопределенных макросов, обеспечивающих информацию о текущем модуле. Это такие макросы, как:

- `?FILE` раскладывается в текущее имя файла.
- `?MODULE` раскладывается в текущее имя модуля.
- `?LINE` раскладывается в текущий номер строки.

Управление потоком в Макросах

Внутри определения макроса поддерживаются следующие директивы. Вы можете использовать их для направления потока управления внутри макроса:

`-undef(Macro)`. Разопределяет макрос; после этого вы не можете вызывать макрос.

`-ifdef(Macro)`. Вычисляет следующие строки только если Macro был определен.

`-ifndef(Macro)`. Вычисляет следующие строки только если Macro не определен.

`-else`. Допустимо после утверждений `ifdef` или `ifndef`. Если условие было `false`, то утверждения, следующие за `else` будут вычислены.

`-endif`. Отмечает конец утверждения `ifdef` или `ifndef`.

Условные макросы должны быть правильно вложены. Условно они группируются следующим образом:

```
-ifdef(debug).  
-define(...).  
-else.
```

```
-define(...).  
-endif.
```

Мы можем использовать эти макросы для определения макроса TRACE. Например:

```
-module(m1).  
-export([start/0]).  
  
-ifdef(debug).  
-define(TRACE(X), io:format("TRACE ~p:~p ~p~n", [?MODULE, ?LINE, X])).  
-else.  
-define(TRACE(X), void).  
-endif. %% MISCELLANEOUS SHORT TOPICS 110  
  
start() -> loop(5).  
loop(0) ->  
    void;  
loop(N) ->  
    ?TRACE(N),  
    loop(N-1).
```

Примечание: `io:format(String, [Args])` печатает переменные в `[Args]` в оболочке Эрланга в соответствии с форматирующими информацией в `String`. Форматирующие коды представлены символом `~`. `~p` является аббревиатурой для *pretty print* (достаточно для печати), а `~n` создает новую строку.⁵

Для компиляции с использованием включенного и выключенного макроса трассировки мы можем использовать дополнительные аргументы в `c/2` следующим образом:

```
1> c(m1, {d, debug}).  
{ok,m1}  
2> m1:start().  
TRACE m1:15 5  
TRACE m1:15 4  
TRACE m1:15 3  
TRACE m1:15 2  
TRACE m1:15 1  
void
```

`c(m1, Options)` обеспечивает способ помещения опций в компилятор. `{d, debug}` устанавливает флаг отладки в `true`, что позволяет распознать его в секции `ifdef(debug)` определения макроса.

Когда макрос выключен, макрос трассировки просто раскладывается в атом `void`.

Такой выбор названия не имеет никакого значения; это просто напоминание для меня, что значение макроса нигде не используется.

Оператор сопоставления в шаблонах

Предположим, что мы имеем некоторый код, такой как:

```
func1([{tag1, A, B}|T]) ->
...
... f(..., {tag1, A, B}, ...)
...
```

В строке 1 мы шаблонно сопоставляем терм `{tag1, A, B}`, а в строке 3 мы вызываем `f` с аргументом, которым является `{tag1, A, B}`. Когда мы делаем это, система пересобирает терм `{tag1, A, B}`. Гораздо более эффективным и менее подверженным ошибкам способом сделать это является присвоение шаблона временной переменной `Z` и помещение ее в `f`, например:

```
func1([{tag1, A, B}=Z|T]) ->
...
... f(... Z, ...)
...
```

Оператор сопоставления может быть использован в любой точке шаблона, так если мы имеем два терма, которые требуется пересобрать, как в следующем коде:

```
func1([{tag, {one, A}, B}|T]) ->
...
... f(..., {tag, {one,A}, B}, ...),
... g(..., {one, A}), ...
...
```

То мы можем представить две новые переменные `Z1` и `Z2` и написать следующее:

```
func1([{tag, {one, A}=Z1, B}=Z2|T]) ->
...,,
... f(..., Z2, ...),
... g(..., Z1, ...),
...
```

Числа

Числа в Эрланге являются либо целыми, либо действительными.

Целые числа

Целочисленная арифметика является точной, а количество цифр, которые могут быть представлены в целом числе ограничивается только доступным объемом памяти.

Целые числа записываются одним из трех различных синтаксисов:

1. **Обычный синтаксис:** Здесь целые числа записаны так, как вы ожидаете. Например, `12`, `12375` и `-23427` являются целыми числами.
2. **Целое по основанию K:** Целые числа по основанию, отличному от десяти записываются с помощью синтаксиса `K#Digits`; то есть мы можем написать двоичное число как `2#00101010` или шестнадцатиричное число как `16#af6bfa23`. Для оснований больше, чем десять символы `abc...` (или `ABC...`) представляют числа `10`, `11`, `12` и так далее. Наибольшим основанием, которое мы можем представить этим способом есть основание `36`.
3. **\$ синтаксис:** Синтаксис `$C` представляет целочисленный код для ASCII-символа `C`. То есть `$a` является сокращением для `97`, `$1` - для `49` и так далее.

Непосредственно после `$` мы можем также использовать любую управляемую последовательность, описанную на рис. 5.1. Таким образом, `$\n` это 10, `$\\^c` это 3 и так далее.

Несколько примеров целых чисел:

```
0 -65 2#010001110 -8#377 16#FE34 36#wow
```

(Их значениями являются: 0, -65, 142, -255, 65076, 65076 и 42368 соответственно.)

Действительные числа

Числа с плавающей точкой имеют пять частей: необязательный знак, целая часть числа, десятичная точка, дробная часть и необязательная часть экспоненты.

Несколько примеров действительных чисел:

```
1.0 3.14159 -2.3e+6 23.56E-27
```

После разбора, числа с плавающей точкой внутренне представляются в 64-битном формате IEEE 754. Реальные числа в диапазоне от `-10323` до `10308` могут быть представлены действительными числами Эрланга.

Старшинство операторов

Таблица 5.2 показывает все операторы Эрланга в порядке убывания приоритета,

вместе с их ассоциативностью. Старшинство операторов и ассоциативность используется для определения порядка вычисления в бесскобочных выражениях.

Выражения с высоким приоритетом (выше в таблице) вычисляются первыми, а затем вычисляются выражения с более низким приоритетом. Таким образом, например, для вычисления $3+4*5+6$ мы сначала вычислим подвыражение $4*5$, так как $*$ выше в таблице, чем $+$. Затем мы вычисляем $3+20+6$. Так как $+$ является лево-ассоциативным оператором, мы рассматриваем его как $(3+20)+6$, поэтому мы вначале вычисляем $3+20$, получив 23 ; в заключение мы вычисляем $23+6$.

В полной скобочной форме $3+4*5+6$ представляется как $((3+(4*5))+6)$. Как и во всех остальных языках программирования, лучше использовать скобки для обозначения области действия, чем полагаться на правила приоритета.

Операторы	Ассоциативность
:	
#	
(унарный) $+$, (унарный) $-$, <code>bnot</code> , <code>not</code>	
$/$, $*$, <code>div</code> , <code>rem</code> , <code>band</code> , <code>and</code>	Лево-ассоциативный
$+$, $-$, <code>bor</code> , <code>bxor</code> , <code>bsl</code> , <code>bsr</code> , <code>or</code> , <code>xor</code>	Лево-ассоциативный
$++$, $--$	Право-ассоциативный
$==$, $/=$, $=<$, $<$, $>=$, $>$, $=:=$, $=/=$	
<code>andalso</code>	
<code>orelse</code>	

Таблица 5.2. Старшинство операторов.

Словарь процесса

Каждый процесс в Эрланге имеет свое собственное персональное хранилище данных, называемое словарем процесса. Словарь процесса является ассоциативным массивом (в других языках это может называться картой, хеш-картой или хеш-таблицей), состоящим из набора ключей и значений. Каждый ключ имеет только одно значение.

Словарем можно манипулировать с использованием следующих BIFов:

```
@spec put(Key, Value) -> OldValue.
```

Добавить ассоциацию Ключ-Значение в словарь процесса. Значением `put` является `OldValue`, которое было предыдущим значением, ассоциированным с `Key`. Если предыдущего значения не было, возвращается атом `undefined`.

`@spec get(Key) -> Value.` Просмотр значения `Key`. Если есть соответствующая ассоциация Key-Value в словаре, то возвращается `Value`; иначе возвращается атом `undefined`.

`@spec get() -> [{Key,Value}]`. Возвращение словаря полностью, как списка кортежей `{Key, Value}`.

`@spec get_keys(Value) -> [Key]`. Возвращает список ключей, который имеют значение `Value` в словаре.

`@spec erase(Key) -> Value`. Возвращает значение, ассоциированное с `Key`, или атом `undefined` если нет ассоцииированной величины. В завершение значение, ассоциированное с `Key` стирается.

`@spec erase() -> [{Key,Value}]`. Стирает словарь процесса полностью.

Возвращаемым значением является список кортежей `{Key, Value}`, представляющий состояние словаря перед тем, как он был очищен.

Например:

```
1> erase().  
[]  
2> put(x, 20).  
undefined  
3> get(x).  
20  
4> get(y).  
undefined  
5> put(y, 40).  
undefined  
6> get(y).  
40  
7> get().  
[{y,40},{x,20}]  
8> erase(x).  
20  
9> get().  
[{y,40}]
```

Как вы можете видеть, переменные в словаре процесса ведут себя во многом как обычные переменные в императивных языках программирования. Если вы используете словарь процесса, ваш код более не является свободным от побочных эффектов, а все преимущества использования неразрушаемых переменных, которые мы обсуждали в разделе 2.6, *Переменные, которые не меняются, не применяются*. По этой причине вы

должны использовать словарь процесса очень осторожно.

Примечание: Я редко использую словарь процесса. Использование словаря процесса может внести тонкие ошибки в вашу программу и сделать ее сложной для отладки. Одной из форм использования, которую я одобряю, является использование словаря процесса для хранения "однократно записанных" переменных. Если ключ принимает значение ровно один раз и не меняет значение, то сохранение его в словаре процесса иногда допустимо.

Ссылки

Ссылки являются глобально уникальными термами Эрланга. Они создаются с помощью BIF `erlang:make_ref()`. Ссылки удобны для создания уникальных тэгов, которые могут быть включены в данные, а затем на более позднем этапе сравнены на равенство. Например, система баг-трекинга может добавлять ссылку на каждый новый отчет об ошибке для того, чтобы получить его уникальный идентификатор.

Упрощенные булевые выражения

Упрощенные булевые выражения - это булевые выражения, аргументы которых вычисляются только если это необходимо.

Существует два упрощенных булевых выражения:

`Expr1 orelse Expr2` Здесь сначала вычисляется `Expr1`. Если `Expr1` вычисляется как истина, то `Expr2` не вычисляется. Если `Expr1` вычислена как ложь, `Expr2` вычисляется.

`Expr1 andalso Expr2` Здесь сначала вычисляется `Expr1`. Если `Expr1` истинно, то `Expr2` вычисляется. Если `Expr1` ложно, `Expr2` не вычисляется.

Примечание: В соответствующих булевых выражениях (`A or B`; `A and B`) оба аргумента вычисляются всегда, даже если правильное значение выражения может быть определено вычислением только первого выражения.

Сравнение термов

Существует восемь возможных операций сравнения термов, показанных в таблице 5.3.

Для целей сравнения определен общий порядок по всем термам. Определено так, что следующее выражение истинно:

number < atom < reference < fun < port < pid < tuple < list < binary

Что это означает? Это означает, что, например, число (любое число) по определению

меньше атома (любого атома), что кортеж больше, чем атом и так далее. (Заметьте, что в целях сравнения, порты и PIDы включены в этот список. Мы поговорим об этом позже.)

Имеющийся общий порядок по всем термам означает, что мы можем отсортировать список любого типа и создать эффективные процедуры доступа на основе порядка сортировки ключей.

Все операторы сравнения термов, за исключением `=:=` и `=/=` ведут себя следующим образом, если их аргументы числа:

Если один аргумент целое, а другой - действительное число, то целое конвертируется в действительное перед выполнением сравнения.

Если оба аргумента целые, или оба аргумента действительные числа, то аргументы используются "как есть", т.е. без преобразования.

Вы должны также быть очень внимательными при использовании `==` (особенно если вы C или Java программист). В 99 из 100 случаев вы должны использовать `=:=`. `==` полезен только для сравнения целых и действительных чисел. `=:=` проверяет, являются ли два терма идентичными⁶. Если вы сомневаетесь, используйте `=:=`, и будьте подозрительными, если увидите `==`. Заметьте, что похожие комментарии применимы к использованию `/=` и `=/=`, где `/=` означает "не равно", а `=/=` означает "не идентично".

Примечание: В большинстве библиотечного и опубликованного кода вы увидите `==`, использованный когда должен быть оператор `=:=`. К счастью, такая ошибка не часто приводит в результате к неверной программе, поскольку если аргументы `==` не содержат действительных чисел, то поведение этих двух операторов то же самое.

Оператор	Значение
<code>X > Y</code>	X больше, чем Y
<code>X < Y</code>	X меньше, чем Y
<code>X <= Y</code>	X равен, либо меньше, чем Y
<code>X >= Y</code>	X равен, либо больше, чем Y
<code>X == Y</code>	X равен Y
<code>X /= Y</code>	X не равен Y
<code>X =:= Y</code>	X идентичен Y
<code>X =/= Y</code>	X не идентичен Y

Таблица 5.3. Сравнение термов.

Вы должны также знать, что сопоставление условия функции всегда подразумевает точное соответствие шаблона, так что если вы определите функцию `F = fun(12) -> ... end`, то попытка вычислить `F(12)` не удастся.

Подчеркнутые переменные

Есть еще одна вещь, которую надо сказать о переменных. Специальный синтаксис `_VarName` используется для обычной, а не для анонимной переменной. Обычно компилятор генерирует предупреждение, если переменная используется только однажды в условии, так как обычно это является признаком ошибки. Если переменная используется только один раз, но начинается с подчеркивания, предупреждающее сообщение сгенерировано не будет.

Так как `_Var` является нормальной переменной, могут произойти очень трудноуловимые ошибки, вызванные забыванием этого и использовании ее как шаблона "не беспокоиться". В сложном шаблонном сопоставлении может быть трудно заметить, например, что `_Int` повторяется, хотя не должна, приводя к неудаче сопоставления шаблона.

Есть два основных порядка использования подчеркнутых переменных:

Именовать переменные, которые мы не намерены использовать. То есть написание `open(File, _Mode)` делает программу более читабельной, чем написание `open(File, _)`.

В целях отладки. Например, предположим мы пишем следующее:

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    io:format("Q = \~p\~n" , [Q]),
    P.
```

Этот код компилируется без сообщения об ошибке.

Теперь закомментируем утверждение `format`:

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    %% io:format("Q = \~p\~n" , [Q]),
    P.
```

Если мы это откомпилируем, компилятор выдаст предупреждение, что переменная `Q` не используется.

Если мы перепишем функцию следующим образом:

```
some_func(X) ->
{P, _Q} = some_other_func(X),
io:format("_Q = ~p~n" , [_Q]),
P.
```

то мы можем комментировать утверждение `format` и компилятор не будет жаловаться.

Теперь мы фактически прошли через последовательный Эрланг. Мы не упомянули о некоторых небольших разделах, но мы вернемся к ним, когда столкнемся с ними в прикладных разделах.

В следующей главе мы рассмотрим, как компилировать и запускать ваши программы различными способами.

Глава 6. Компилярование и запуск ваших программ

В предыдущих главах, мы мало говорили о компиляции и запуске программ - мы просто использовали оболочку Эрланг. Это вполне нормально для небольших примеров. Но по мере роста сложности ваших программ, вы, несомненно, захотите как-то автоматизировать этот процесс, чтобы упростить себе жизнь. Вот здесь и появляются `make`-файлы.

Существует три разных способа запуска ваших программ. И в этой главе, мы рассмотрим их все, так что вы сможете выбрать тот, который наиболее подходит к вашей ситуации.

Иногда могут происходить сбои: `make`-файлы могут не срабатывать, переменные окружения быть неправильными, также как и пути поиска файлов. Мы поможем вам разобраться с подобными проблемами, подсказав, что делать, если что-то пошло не так.

6.1 Запуск и остановка Эрланг оболочки (shell)

На UNIX системах (Включая Mac OS X) вы можете запустить оболочку Эрланга из командной строки консоли:

```
$ erl
```

```
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]
Eshell V5.5.1 (abort with ^G)
1>
```

В системе Microsoft Windows вам надо кликнуть на иконке Эрланга.

Простейшим способом остановить систему является нажатие Ctrl+C (в Виндоус - Ctrl+Break) и далее - A , как в следующем примере:

```
BREAK: (a)abort (c)ontinue (p)roc info (i)nfo (l)oaded
      (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

Также, вместо этого, вы можете выполнить в оболочке Эрланга (или в программе) инструкцию `erlang:halt()` , что приведет к тому-же результату.

`erlang:halt()` - это BIF которая немедленно останавливает оболочку Эрланга и именно этим способом я и пользуюсь в большинстве случаев. Но, тем не менее, в этом способе есть определенное неудобство. Если вы запустили большое приложение работающее с базами данных и просто остановите всю систему, то при следующем запуске вам придется проходить через процесс восстановления после ошибки. Поэтому лучше останавливать систему более аккуратным способом.

Для контролируемой остановки, если оболочка реагирует на команды, вы можете набрать:

```
1> q().
ok
$
```

Тогда будут корректно закрыты все открытые файлы, остановлены базы данных (если они запущены) и закрыты все OTP приложения в установленном порядке. Команда `q()` это другое имя для команды `init:stop()`.

Если ни один из этих методов не работает, прочитайте раздел 6.6 *Как выбраться из неприятностей*.

6.2. Изменение окружения

Когда вы начинали программировать на Эрланге, вы, возможно, складывали все файлы и все модули в одну директорию из которой и стартовали сам Эрланг. В этом

случае Эрланг без проблем мог найти ваш код. Но по мере роста сложности ваших приложений, вы наверняка захотите разделить его в управляемые куски и разместить их в различных директориях. А если вы будете включать в ваш проект внешний код, то он будет иметь свою собственную структуру директорий.

Установка пути поиска для загрузки кода

Система исполнения приложений Эрланг использует механизм автозагрузки кода. Чтобы он работал корректно вы должны правильно установить несколько путей поиска, чтобы находилась именно правильная версия вашего кода.

Механизм загрузки кода на самом деле также написан на Эрланг - подробнее мы поговорим об этом в разделе E.4 *Динамическая загрузка кода*. И загрузка кода осуществляется , как это называют, "по требованию".

Когда система пытается вызвать функцию в модуле, который еще не был загружен, возникает исключение, и система пытается найти файл объектного кода пропущенного модуля. Если, например, этот пропущенный модуль называется *myMissingModule*, то загрузчик кода, первым делом, будет пытаться найти файл с именем *MyMissingModule.beam* во всех директориях, которые входят в текущий путь загрузки кода. Поиск останавливается на первом таком найденном файле и объектный код из этого файла загружается в систему.

Вы можете узнать текущий путь загрузки кода запустив Эрланг и набрав команду `code:get_path()`. Вот пример ее работы:

```
code:get_path().  
[".",  
 "/usr/local/lib/erlang/lib/kernel-2.11.3/ebin",  
 "/usr/local/lib/erlang/lib/stdlib-1.14.3/ebin",  
 "/usr/local/lib/erlang/lib/xmerl-1.1/ebin",  
 "/usr/local/lib/erlang/lib/webtool-0.8.3/ebin",  
 "/usr/local/lib/erlang/lib/typer-0.1.0/ebin",  
 "/usr/local/lib/erlang/lib/tv-2.1.3/ebin",  
 "/usr/local/lib/erlang/lib/tools-2.5.3/ebin",  
 "/usr/local/lib/erlang/lib/toolbar-1.3/ebin",  
 "/usr/local/lib/erlang/lib/syntax_tools-1.5.2/ebin",  
 ...]
```

Следующие две функции наиболее часто используются для работы с путем загрузки кода:

```
@spec code:add_patha(Dir) => true | {error, bad_directory}
```

Добавляет новую директорию Dir в начало пути загрузки кода.

```
@spec code:add_pathz(Dir) => true | {error, bad_directory}
```

Добавляет новую директорию Dir в конец пути загрузки кода.

Часто это совершенно не важно какую из них использовать. Но надо следить за случаями, когда `add_patha` и `add_pathz` приводят к различным результатам. Если вы подозреваете, что загрузился не тот модуль, то вы можете набрать `code:all_loaded()` (которая выводит все загруженные модули) или `code:clash()` ("столкновение, коллизия") чтобы разобраться, что же именно произошло не так.

Есть, также, еще несколько процедур в модуле `code` для работы с путем загрузки, но возможно они вам никогда не понадобятся, если только вы не станете писать весьма странные системные программы.

Обычной практикой является размещение всех этих команд в файле `.erlang` в вашей домашней директории. Либо же вы можете запускать Эрланг командой следующего вида:

```
> erl -pa Dir1 -pa Dir2 ... -pz DirK1 -pz DirK2
```

где флаг `-pa DirX` добавляет директорию `DirX` в начало пути поиска кода, а `-pz DirY` добавляет директорию `DirY` в конец пути поиска кода.

Выполнение набора команд при старте системы Эрланг

Мы уже познакомились, как можно установить путь поиска кода через команды в файле `.erlang` в вашей домашней директории. Но на самом деле вы можете поместить в этот файл любые команды Эрланг и, когда вы запустите Эрланг, он первым делом прочитает и выполнит все команды из этого файла.

Предположим мой файл `.erlang` выглядит следующим образом:

```
io:format("Running Erlang ~n").
code:add_patha(".").
code:add_pathz("/home/joe/2005/erl/lib-supported").
code:add_pathz("/home/joe/bin").
```

Тогда, при старте системы я увижу следующий вывод на консоль:

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]
Running Erlang
```

```
Eshell V5.5.1 (abort with ^G)
```

```
1>
```

Если в той директории, откуда стартует Эрланг также имеется файл `.erlang` то исполняться будет именно он, а не файл в вашей домашней директории. Таким образом вы можете настроить поведение Эрланга в зависимости от того, где он начал свою работу. Это может быть полезно для специализированных приложений. В таком случае, вероятно, будет неплохой идеей добавить в такой файл несколько команд печати соответствующих сообщений, поскольку, в противном случае, вы можете забыть, что это локальный файл начальной настройки системы, что может привести к затруднениям в работе.

Подсказка: В некоторых системах, иногда бывает не совсем очевидно, а где именно находится ваша домашняя директория. Чтобы определить, что думает Эрланг, где она находится, сделайте следующее:

```
1> init:get_argument(home).  
{ok,[["/home/joe"]]}
```

Откуда мы можем понять, что Эрланг думает, что нашей домашней директорией является `/home/joe`.

6.3. Различные способы запустить вашу программу

Программы Эрланга хранятся в модулях. как только вы написали вашу программу вы должны ее скомпилировать перед тем как запускать ее. Однако вы можете запустить вашу программу без компиляции запустив `escript`.

В следующих разделах мы покажем как скомпилировать и запустить несколько программ разными способами. Программы немного различаются, равно как и способы их запуска и остановки.

Первая программа `hello.erl` просто печатает фразу "Hello world!". Она не будет запускать или останавливать систему Эрланг и ей не нужен доступ к аргументам в командной строке. Но наша вторая программа `fac`, наоборот будет нуждаться в доступе к параметру командной строки при ее запуске.

Вот наша простейшая программа, сохраненная в файле `hello.erl`. Она печатает строку "Hello World" с последующим переходом на новую строку (символ `\n` интерпретируется как новая строка в модулях Эрланга `io` и `io_lib`).

[Скачать `hello.erl`](#)

```
-module(hello).  
-export([start/0]).  
  
start() ->  
    io:format("Hello world~n").
```

Давайте скомпилируем и запустим эту программу тремя разными способами.

Компиляция и запуск в оболочке Эрланг (shell)

```
$ erl  
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]  
  
Eshell V5.5.1 (abort with ^G)  
1> c(hello).  
{ok,hello}  
2> hello:start().  
Hello world  
ok
```

Компиляция и запуск из командной строки

```
$ erlc hello.erl  
$ erl -noshell -s hello start -s init stop  
Hello world  
$
```

Быстрое исполнение

Часто возникает необходимость выполнить некоторую функцию Эрланга из командной строки операционной системы. В этом случае ключ `-eval` с аргументом бывает очень полезен для такого быстрого исполнения.

Приведем пример:

```
erl -eval 'io:format("Memory: ~p~n" , [erlang:memory(total)]).' -noshell  
-s init stop
```

Пользователи Microsoft Windows: Чтобы это сработало, вам надо либо добавить в переменную PATH директорию содержащую исполняемый код системы Эрланг, либо вызывать erlc и erl полным именем (включая двойные кавычки).

Например:

```
"C:\Program Files\erl5.5.3\bin\erlc.exe" hello.erl
```

..

Первая строка приведенная выше `erlc hello.erl` компилирует файл `hello.erl` в файл объектного кода `hello.beam`. Вторая команда возможна в трех различных вариантах:

`-noshell`

Запускает систему Эрланг без интерактивной оболочки (то есть вы не увидите стандартной надписи при запуске Эрланга).

`-s hello start`

Запускается функция `hello:start()`.

Примечание: При использовании `-s Module` сам `Module` должен быть уже скомпилированным.

`-s init stop`

Когда закончится исполнение `apply(hello, start, [])` система автоматически выполнит функцию `init:stop()`.

Команда `erl -noshell ...` может использоваться в командных файлах интерпретатора команд ОС (*shell scripts*), так что типичным случаем является создание командных файлов, которые устанавливают пути поиска (с помощью `-pa Directory`) и запускают программу.

В нашем примере мы использовали две `-s` команды. Число их в командной строке не ограничено. Каждая `-s` команда будет выполнена с помощью функции `apply`, а когда она закончится, начнется выполнение следующей команды.

Вот пример запуска `hello.erl`

Скачать [hello.sh](#)

```
#!/bin/sh
erl -noshell -pa /home/joe/2006/book/JAERANG/Book/code\
-s hello start -s init stop
```

Примечание: Этот скрипт нуждается в абсолютном пути до директории, где находится файл `hello.beam`. Так что, хотя он и работает на моей машине, вам придется подредактировать его, чтобы он работал на вашей.

Чтобы запустить этот скрипт вы должны установить ему соответственно атрибуты командой `chmod` (только один раз) и тогда, будет можно его запускать:

```
$ chmod u+x hello.sh  
$ ./hello.sh  
Hello world  
$
```

Примечание: В Microsoft Windows, прием `#!` не работает. Там придется создать `.bat` файл и использовать полный путь к исполняемой части системы Эрланг, если переменная `PATH` для нее не установлена соответственно.

Типичный скриптовый файл для Microsoft Windows может выглядеть приблизительно так:

[Скачать `hello.bat`](#)

```
"C:\Program Files\erl5.5.3\bin\erl.exe" -noshell -s hello start -s init stc
```

Запуск через Escript

Используя `escript` вы можете запускать свои программы именно как скрипты, без их предварительной компиляции.

Предупреждение: `escript` входит в Эрланг начиная с версии R11B-4 и далее. Если у вас более ранняя версия Эрланга, то вы должны ее обновить до последней версии системы Эрланг.

Для запуска `hello` как скрипта, мы создадим следующий файл:

[Скачать `hello`](#)

```
#!/usr/bin/env escript  
main(_) ->  
io:format("Hello world ~n" ).
```

Экспорт функций во время разработки

Когда вы разрабатываете код, вам может быть немного неудобно все время возвращаться к разделу объявления экспорта функций, только для того, чтобы была возможность запустить их в оболочке Эрланг.

Специальная декларация для компилятора `-compile(export_all)` указывает ему экспортировать все функции в модуле. Это существенно упрощает жизнь пока вы разрабатываете код.

Когда вы закончили разработку кода в этом модуле, вы должны закомментировать декларацию `export_all` и добавить более точные декларации по экспорту функций. На это есть две причины. Во-первых, когда вы придете в следующий раз читать ваш код, вы будете знать, что только важные функции были проэкспортированы наружу, а остальные функции не могут быть вызваны снаружи, так что вы можете их менять, как вам это нужно, сохраняя лишь их интерфейс к проэкспортированным функциям. А во-вторых, компилятор может произвести гораздо более лучший код, когда он точно знает, какие именно функции проэкспортированы из данного модуля.

На UNIX системах мы можем запустить этот файл немедленно и без всякой компиляции:

```
chmod u+x hello
$ ./hello
Hello world
$
```

Примечание: Параметры данного файла должны быть переведены в исполняемые (что в UNIX достигается командой `chmod u+x File`), что нужно сделать только один раз, а не каждый раз при запуске программы.

Программы с аргументами в командной строке

"Hello world" не имеет никаких аргументов. Давайте повторим наше упражнение для программы, которая вычисляет факториалы. Ей потребуется один аргумент.

Во-первых, вот ее код

Скачать `fac.erl`

```
-module(fac).
-export([fac/1]).

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

Мы можем скомпилировать `fac.erl` и запустить его в оболочке Эрланга следующим образом:

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1 (abort with ^G)
```

```
1> c(fac).
{ok,fac}
2> fac:fac(25).
15511210043330985984000000
```

Если же мы хотим запускать эту программу из командной строки, то нам надо ее модифицировать, чтобы она воспринимала оттуда параметр :

[Скачать fac1.erl](#)

```
-module(fac1).
-export([main/1]).  
  
main([A]) ->  
    I = list_to_integer(atom_to_list(A)),  
    F = fac(I),  
    io:format("factorial ~w = ~w~n" ,[I, F]),  
    init:stop().  
  
fac(0) -> 1;  
fac(N) -> N*fac(N-1).
```

Теперь мы можем ее скомпилировать и запустить:

```
$ erlc fac1.erl
$ erl -noshell -s fac1 main 25
factorial 25 = 15511210043330985984000000
```

Примечание: То что функция называется `main` - это не важно. Она может называться как угодно. Важно только чтобы ее имя и имя функции в командной строке совпадали.

Наконец, мы можем запустить ее как скрипт:

[Скачать factorial](#)

```
#!/usr/bin/env escript  
  
main([A]) ->  
    I = list_to_integer(A),  
    F = fac(I),  
    io:format("factorial ~w = ~w ~n" ,[I, F]).  
  
fac(0) -> 1;  
  
fac(N) ->
```

```
N * fac(N-1).
```

Компиляция здесь не нужна, просто запускаем его:

```
$ ./factorial 25
factorial 25 = 15511210043330985984000000
$
```

6.4. автоматическая компиляция в make-файлах

Когда я пишу большую программу, я хочу автоматизировать этот процесс настолько, насколько это вообще возможно. На это есть две причины. Во-первых, в долгосрочной перспективе, это спасает от набивания опять и опять одних и тех-же команд во время многократного тестирования моей программы из множества строк и я не хочу чтобы мои пальцы у меня отвалились от всего этого.

Во-вторых, я часто откладываю текущий проект и вынужден поработать над чем-то еще. Могут пройти месяцы, прежде чем я вернусь к отложенному проекту и я совершенно забываю, как надо обрабатывать в нем код, что не оставляет мало шансов на его спасение!

`make` - это утилита, которая автоматизирует мою работу. Я использую ее для компиляции и распределения моего Эрланг - кода. Большинство моих make-файлов чрезвычайно просты и у меня есть для них готовые шаблоны, которые решают абсолютное большинство моих проблем.

Я не буду рассказывать о make-файлах в общем (см., например: <http://ru.wikipedia.org/wiki/Make>). Вместо этого я покажу их форму, которая кажется мне полезной для компиляции Эрланг программ. В частности, мы рассмотрим make-файлы, прилагаемые к данной книге, что значит, что вы будете понимать их и уметь писать, на их основе, собственные make-файлы.

Шаблон make-файла

Вот шаблон, на основе которого я создаю большинство своих make-файлов

Скачать [Makefile.template](#)

```
# Эти строки не меняйте
.SUFFIXES: .erl .beam .yrl

.erl.beam:
```

```
erlc -W $<

.yrl.erl:
    erlc -W $<

ERL = erl -boot start_clean

# Здесь указывается список модулей которые будем компилировать
# Если модули не помещаются в одну строку можно добавить символ \
# в конце строки и продолжить на следующей строке

# Отредактируйте строки которые находятся ниже

MODS = module1 module2 \
       module3 ... special1 ...\
       ...
       moduleN

# The first target in any makefile is the default target.
# If you just type "make" then "make all" is assumed (because
# "all" is the first target in this makefile)

all: compile

compile: ${MODS:%=%.beam} subdirs

## Специальный метод компиляции

special1.beam: special1.erl
    ${ERL} -Dflag1 -W0 special1.erl

## run an application from the makefile

application1: compile
    ${ERL} -pa Dir1 -s application1 start Arg1 Arg2

# the subdirs target compiles any code in
# sub-directories

subdirs:
    cd dir1; make
    cd dir2; make
    ...
    ...

# remove all the code
```

```
clean:  
    rm -rf *.beam erl_crash.dump  
    cd dir1; make clean  
    cd dir2; make clean
```

Данный make-файл начинается с правил компиляции модулей Эрланга и файлов с расширением `.yrl` (эти файлы содержат определения для программы Эрланг парсер-генератор)

Важной частью make-файла является следующая строка:

```
MODS = module1 module2
```

Это список всех Эрланг-модулей, которые я хочу скомпилировать.

Каждый модуль из строки MODS будет скомпилирован командой `erlc Mod.erl`. Некоторые модули могут требовать специального обхождения с ними (например модуль `special1` в файле-шаблоне), поэтому для них есть отдельное правило обработки.

Внутри make-файла имеется много целей. Цель - это символьно-числовая строка, начинающаяся в первой позиции строки make-файла и оканчивающаяся двоеточием `:`. В нашем make-файле шаблоне `all`, `compile` и `special.beam` все являются целями. Чтобы выполнить make-файл вы даете следующую команду в консоли:

```
$ make [Target]
```

Параметр *Target* - не обязательный. Если его нет, тогда подразумевается первая цель в файле. В нашем примере - это цель `all`, если другой цели не было задано в командной строке.

Если я хочу перестроить мою программу и запустить `application1`, тогда я отдаю команду `make application1`. Если я хочу чтобы это было поведением по умолчанию, вызываемым только по команде `make`, то мне надо передвинуть строки определяющие цель `application1` так, чтобы они стали первой целью в make-файле.

Цель `clean` удаляет все скомпилированные объектные коды Эрланга и файл `erl_crash.dump`. Этот файл содержит информацию, которая может помочь отладить ваше приложение. Более детально данная тема рассмотрена в разделе 6.10 *Аварийный сброс системы (Crush Dump)*.

Специализация шаблона Make-файла

Я не сторонник беспорядка в моих программах, поэтому я обычно начинаю с шаблона make-файла и удаляю из него все строки не имеющие отношения к моему приложению. Это дает мне make-файл который короче и гораздо более понятный при его прочтении. Но, с другой стороны, вы можете иметь обобщенный, универсальный make-файл, включаемый во все make-файлы, поведение которого определяется их конкретными, специфическими переменными.

Результатом моего, вышеуказанного, процесса, может быть, например, нижеуказанный make-файл:

```
.SUFFIXES: .erl .beam

.erl.beam:
    erlc -W $<

ERL = erl -boot start_clean

MODS = module1 module2 module3

all: compile
    ${ERL} -pa '/home/joe/.../this/dir' -s module1 start

compile:
    ${MODS:%=%.beam}

clean:
    rm -rf *.beam erl_crash.dump
```

6.5. Редактирование командами в оболочке Эрланга

Оболочка Эрланга содержит встроенный строковый редактор. Его команды являются подмножеством команд редактирования строчек используемых в популярном редакторе emacs. Предыдущие строки могут быть вызваны и отредактированы несколькими командными символами. Они приведены далее (учтите что *Key* означает, что вы должны нажать *Ctrl+Key*):

Команда	Описание
^A	Начало строки
^E	Конец строки
^F или стрелка вправо	Вперед на символ

<code>^B</code> или стрелка влево	Назад на символ
<code>^P</code> или стрелка вверх	Предыдущая строка
<code>^N</code> или стрелка вниз	Следующая строка
<code>^T</code>	Поменять последние два символа
Табуляция	Попытаться дописать имя текущего модуля или функции

6.6. Как выбраться из неприятностей

Эрланг иногда бывает трудно остановить. Перечислим здесь некоторые причины этого:

- Оболочка Эрланга не отвечает
- Обработка команды `Ctrl+C` была выключена
- Эрланг был запущен с флагом `-detached` так, что вы можете не знать, что он работает
- Эрланг был запущен с опцией `-heart Cmd`. Эта опция заставляет мониторинг процессов ОС следить за ОС-процессом Эрланга. И если Эрланг-процесс в ОС умирает, то выполнить команду `Cmd`. Часто, при этом, `Cmd` просто перезапускает систему Эрланг. Это один из трюков, который используется для создания отказоустойчивых узлов Эрланга. Если вдруг слетит сам Эрланг (что, вообще говоря, не должно происходить), он просто будет перезапущен. Хитростью при остановке Эрланга теперь является найти сначала тот замеряющий пульс Эрланг-процесс (используйте `ps` на UNIX-подобных машинах и Менеджер Задач (Task Manager) в Microsoft Windows и убить его перед тем как убивать процесс Эрланга в данной ОС).
- Что-то может пойти совсем не так и оставить вас с непривязанным зомби-процессом Эрланга.

6.7. Когда что-то пошло не так

В этом разделе перечисляются некоторые общие проблемы и варианты их решения.

Неопределенный (Потерянный) код

Если вы пытаетесь запустить код в модуле, который загрузчик не может найти (поскольку заданный ему путь поиска некорректен) вы встретитесь с `undef`

(неопределен) - сообщением об ошибке. Вот его пример:

```
1> glurk:oops(1,23).
** exited: {undef,[{glurk,oops,[1,23]}, {erl_eval,do_apply,5}, {shell,exprs,6}, {shell,eval_loop,3}]}
```

На самом деле, здесь просто не существует модуля с именем `glurk`, но не это важно. Вы должны сконцентрироваться на рассмотрении сообщения об ошибке. Оно говорит нам, что система Эрланг пыталась вызвать функцию `oops` с аргументами 1 и 23 из модуля `glurk`. Следовательно, возможны четыре варианта, того что произошло при этом.

- Возможно действительно не существует модуль `glurk`.
- Возможно его имя было указано слегка неверно (с опечаткой).

Ктонибудь видел мою точку с запятой?

Если вы забыли поставить точку с запятой между вариантами вызова вашей функции, или поставили туда точку, вместо этого, то у вас будут большие неприятности.

Если вы определили функцию `foo/2` в строке 1234 своего модуля `bar` и поставили точку вместо точки с запятой, компилятор скажет вам:

```
bar.erl:1234 function foo/2 already defined.
```

Не делайте этого. Убедитесь, что ваши варианты функции всегда отделены друг от друга точкой с запятой.

- Модуль `glurk` существует, но он не был скомпилирован. Система ищет файл с именем `glurk.beam` в директориях указанных ей в пути поиска кода.
- Модуль `glurk` существует и он был откомпилирован, но директория, в которой находится `glurk.beam` не входит в путь поиска кода. Чтобы исправить эту ошибку вам, возможно, потребуется изменить путь поиска кода. Как это сделать мы рассмотрим чуть позже.
- Существует несколько версий `glurk` в директориях поиска кода и мы выбрали не тот что нужно. Это редкая ошибка, но такое тоже возможно. Если вы подозреваете, что произошло именно это вы можете запустить функцию `code:clash()`, которая сообщает обо всех повторяющихся модулях в директориях входящих в путь поиска кода.

Мой маке-файл ничего не создает

Что может, вообще, случиться с маке-файлом? Ну, на самом деле, много чего. Но эта книга не про то как с ними работать, так что я ограничусь только самыми распространенными ошибками, связанными с ними.

Вот две самые частые ошибки с которыми я сталкивался:

- *Пробелы в маке-файле*: Make-файлы крайне привередливы. Хотя вы и не можете их видеть, но все, связанные с предыдущей, строки должны начинаться с символа табуляции (за исключением продолжения строк, где на предыдущей строке должен стоять в конце символ `\`). Если вдруг там окажутся пробелы, то маке-файл очень сильно будет смущен и вы отправитесь на поиски ошибки.
- *Пропущенный Эрланг-файл*: Если отсутствует один из модулей указанный в цели маке-файла `MODS`, вы получите сообщение об ошибке. Для примера предположим, что `MODS` содержит модуль с именем `glurk`, но файла с именем `glurk.erl` нет в директории содержащей код. В этом случае маке выдаст следующую ошибку:

```
$ make
make: *** No rule to make target 'glurk.beam',
      needed by 'compile'. Stop.
```

Возможно, также, что такой модуль присутствует, но, просто, его имя было указано некорректно в маке-файле.

Оболочка Эрланга не отвечает

Если оболочка не отвечает на команды, то могла произойти масса причин для этого. Сам процесс оболочки мог погибнуть, или вы могли запустить команду, которая никогда не закончится. Вы также могли забыть закрывающие кавычки или забыть набрать точка-возврат-каретки в конце вашей команды.

Но независимо от причин, вы можете прервать вашу оболочку Эрланга нажатием клавиш `Ctrl+G` и проследовать за следующим примером:

```
1>receive foo -> true end.
^G
User switch command

--> h
  c [nn] - connect to job`
  i [nn] - interrupt job
  k [nn] - kill job
  j - list all jobs
  s - start local shell
  r [node] - start remote shell
```

```
q - quit erlang
? | h - this message`  
  
--> j
1* {shell,start,[init]}  
  
--> s
--> j
1 {shell,start,[init]}
2* {shell,start,[]}  
  
--> c 2
Eshell V5.5.1 (abort with ^G)
1> init:stop().
ok
2> $
```

1. Здесь я сказал оболочке принять сообщение `foo`. Но поскольку никто никогда не собирается посыпать оболочке это сообщение, она входит в бесконечное ожидание этого сообщения. Я нажимаю `Ctrl+G`.
2. Система входит в режим контроля работ (*Job Control Mode* или *JCL*). Тут я никогда не могу вспомнить его команды, поэтому я набираю `h` чтобы вызвать подсказку.
3. Я набираю `j` чтобы вывести весь список работ. Работа номер 1 отмечена символом звездочки, что означает, что это оболочка Эрланга по умолчанию. Все прочие команды используют именно ее, если только им не указан параметр вида `[nn]`.
4. Я набираю `s` чтобы запустить новую оболочку и опять набираю `j`. На этот раз я вижу что уже есть две оболочки с номерами `1` и `2` и оболочка с номером `2` стала оболочкой по умолчанию.
5. Я набираю `2` что подсоединяет меня к только что запущенной оболочке `2`, в которой я останавливаю систему Эрланг.

Как видите у вас может быть множество оболочек в системе Эрланг, в которых можно набирать команды и переключаться между ними нажимая `Ctrl+G`. Вы даже можете запустить оболочку на удаленном Эрланг-узле с помощью команды `r` с соответствующим параметром.

6.8. ВЫЗОВ ПОМОЩИ

В UNIX системах это делается так:

```
$ erl -man erl
NAME
erl - The Erlang Emulator

DESCRIPTION
The erl program starts the Erlang runtime system.
The exact details (e.g. whether erl is a script
or a program and which other programs it calls) are system-dependent.
...
```

Вы также можете получить справку по отдельным модулям:

```
$ erl -man lists
MODULE
lists - List Processing Functions

DESCRIPTION
This module contains functions for list processing.
The functions are organized in two groups:
...
```

Примечание: На UNIX системах страницы справки по умолчанию не установлены. Если команда `erl -man` не работает, то вам нужны страницы справки по Эрланг. Все они имеются в сжатом архиве по адресу <http://www.erlang.org/download.html>. Справочные страницы Эрланга должны быть разархивированы в корневой директории установки Эрланга (обычно это `/usr/local/lib/erlang`).

Документацию по Эрланг также можно загрузить в виде связанного множества HTML файлов. В ОС Microsoft Windows HTML документация устанавливается по умолчанию и доступна в разделе Эрланг в меню Старт данной ОС.

6.9 Настройка окружения

Оболочка Эрланга имеет множество встроенных команд. Вы можете прочитать о них всех с помощью команды оболочки `help()`:

```
1> help().
** shell internal commands **
b() -- display all variable bindings
e(N) -- repeat the expression in query <N>
f() -- forget all variable bindings
```

```
f(X) -- forget the binding of variable X  
h() -- history  
...
```

Все эти команды определены в модуле `shell_default`.

Если вы хотите определить для оболочки свои собственные команды, просто создайте модуль с именем `user_default`. Например:

Скачать [user_default.erl](#)

```
-module(user_default).  
-compile(export_all).  
  
hello() ->  
    "Hello Joe how are you?".  
  
away(Time) ->  
    io:format("Joe is away and will be back in ~w minutes ~n", [Time]).
```

Как только он будет скомпилирован и размещен где-то на пути вашего поиска кода, вы сможете вызывать любую его функцию без указания имени самого модуля:

```
1> hello().  
"Hello Joe how are you?"  
2> away(10).  
Joe is away and will be back in 10 minutes  
ok
```

6.10 Аварийный сброс данных системы (Crash Dump)

Если система Эрланг падает, то она оставляет после этого файл с именем `erl-crash.dump`. Содержимое этого файла может помочь вам разобраться, что же именно пошло не так. Чтобы его проанализировать есть анализатор аварий основанный на веб-интерфейсе. Чтобы его запустить наберите следующую команду:

```
1> webtool:start()**  
WebTool is available at http://localhost:8888/  
Or http://127.0.0.1:8888/  
{ok,<0.34.0>}
```

Потом укажите вашему браузеру адрес <http://localhost:8888> и вы сможете с

удовольствием покопаться в журнале ошибки `(error log)`.

Итак, мы закончили рассмотрение низко-уровневых механизмов системы Эрланг и теперь можем перейти к параллельным программам. С этого момента вы вступаете на незнакомую территорию, но именно тут-то и начинается самое веселье.

1. Я не знаю можно ли запустить escript в Microsoft Windows. Если кто-то знает, как это сделать, напишите мне письмо об этом и я добавлю эту информацию в книгу.
2. Эрланговский парсер-генератор называется `yecc` (Эрланговская версия `yacc` - от "yet another compiler compiler" (еще один компилятор компиляторов)) Смотри руководство по нему в Интернете по адресу:
http://www.erlang.org/contrib/parser_tutorial-1.0.tgz.

Глава 7. Параллельность

Мы понимаем параллельность мира

Глубокое понимание параллельности окружающего нас мира глубоко впечатано в наши мозги. Мы реагируем на внешние раздражители очень быстро при помощи отдела нашего мозга, называемого амигдала. Без такого реагирования мы бы довольно скоро погибли. Наше сознание, по сравнению с подобными реакциями, гораздо медленнее. За то время, пока сама мысль "нажми на тормоз" сформируется в нашей голове, это действие будет нами уже выполнено.

Когда мы ведем автомобиль в насыщенном транспортном потоке, мы ментально отслеживаем местоположение и скорость десятков, если не сотен автомобилей. Все это происходит без участия сознания. Если бы мы не умели это делать, мы бы, вероятно, погибли бы на дороге.

Наш мир параллелен

Если мы хотим писать программы, которые ведут себя подобно другим объектам реального мира, тогда эти программы должны быть параллельными по своей внутренней структуре.

Вот почему мы должны программировать на параллельном языке программирования.

А мы пока чаще программируем приложения для реального мира с помощью последовательных языков. Это излишнее усложнение.

Используйте язык, который был специально разработан для написания параллельных

приложений и разработка параллельных систем станет намного проще и приятнее.

Программы на Эрланге моделируют наш процесс мышления и взаимодействия.

У нас с вами нет общей разделяемой памяти. У меня есть своя память, а у вас - своя. Каждый из нас имеет свои мозги, по одному на брата. Они не соединены вместе. Чтобы изменить вашу память, я должен послать вам сообщение: что-то сказать или, хотя бы, помахать рукой.

Вы слышите, вы смотрите и ваша память изменяется. Но, тем не менее, не задав вам соответствующего вопроса или не видя вашей реакции, я не могу быть уверенным, что вы получили мое сообщение.

Чтобы убедиться в том что другой процесс получил ваше сообщение и изменил свою память, вы должны спросить его об этом (послав ему сообщение). Именно так мы сами и взаимодействуем.

Сью: Привет, Билл, мой номер телефона - 45 67 89 12

Сью: Ты меня понял?

Билл: Конечно, твой номер телефона - 45 67 89 12.

Подобные паттерны взаимодействия очень хорошо нам известны. От самого рождения мы учимся, как взаимодействовать с окружающим миром наблюдая за ним, а также посылая разного рода сообщения и наблюдая за ответной реакцией.

Люди функционируют, как независимые сущности, которые общаются с помощью посылки друг другу сообщений.

Именно так работают процессы в Эрланге и именно так работаем и мы сами, а это значит, что нам будет очень просто понять механизмы работы программ написанных на Эрланге.

Программы на Эрланге состоят из десятков, сотен, а иногда и сотен тысяч маленьких процессов. И все они работают независимо друг от друга. Они общаются между собой путем посылки друг другу сообщений. Каждый процесс имеет свою собственную память. Они ведут себя как некое громадное сорище людей, которые общаются друг с другом.

Это позволяет значительно проще управлять и масштабировать программы на Эрланге. Предположим у нас есть десять людей (процессов) и слишком много работы, которую они должны сделать. Что мы можем сделать в такой ситуации? Позвать на помощь больше людей. А как нам управлять этими группами людей? Это просто - надо

просто рассказать им, всем сразу, их инструкции (широковещательное сообщение от одного ко многим).

Процессы в Эрланге не имеют общей памяти, так что нет никакой необходимости в ее блокировке перед ее использованием. Там где есть замки (блокировки) там будут и потерянные ключи к этим замкам. А что происходит, когда вы теряете свой ключ? Вы паникуете и не знаете, что же вам теперь делать. И то же самое происходит в программных системах, теряются ключи и ваши блокировки заклинивает.

Распределенное программное обеспечение с блокировками и ключами к ним, всегда подвержено этому риску и страдает от этого.

А в эрланге просто нет никаких блокировок и ключей.

Если кто-то умирает, другие люди это замечают.

Если я, находясь в помещении с людьми, вдруг упаду и умру, то кто-то, вероятно, это заметит (по крайней мере, я на это надеюсь). Процессы Эрланга в этом также очень похожи на людей - они могут внезапно умереть. Но в отличии от людей, когда они умирают, они громко выкрикивают на своем последнем вздохе точную причину от чего они умерли.

Представьте себе комнату полную людей. Неожиданно кто-то из них падает и умирает. И в самый момент своей смерти он произносит: " Я умер от сердечного приступа" или "Я умер от разрыва слепого отростка желудка". Также поступают и процессы Эрланга. один процесс, умирая, может сказать "Я умер, потому что меня попросили разделить на ноль". А другой, возможно, скажет "Я умер, потому что меня спросили каков последний элемент в пустом списке".

А теперь, давайте представим, что в нашей комнате полной людей, есть специально назначенные люди, работа которых заключается в том, чтобы позаботиться о трупах. Давайте представим себе двух таких людей, Джейн и Джона. Если умирает Джейн, то Джон решает все проблемы, связанные с ее смертью. А если умирает Джон, то все эти проблемы решает Джейн. То есть, получается, что Джейн и Джон как бы связаны между собой незримым соглашением, в котором говорится, что если один из них умирает, то другой разбирается со всеми проблемами связанными с этой смертью.

Именно так работает механизм ошибок в Эрланге. Процессы в нем могут быть связаны вместе. И если один из них умирает, то другие получат сообщение об ошибке, говорящее о том что этот процесс умер.

Вот, в основном, и все.

Вот так и работают программы на Эрланге.

Итак повторим, то что мы уже узнали:

Эрланг программы состоят из множества процессов. Эти процессы посылают друг другу сообщения

Эти сообщения могут быть и не приняты адресатом и не поняты им. Если вы хотите точно узнать, что ваше сообщение было получено и понято, вы должны послать соответствующее сообщение этому процессу и ждать его ответа.

Пары процессов могут быть связаны вместе. Если один из таких связанных процессов умирает, другому процессу из этой пары будет послано сообщение, содержащее в себе указание на причину смерти первого процесса.

Эту простую модель программирования я называю *параллельно ориентированное программирование*.

В следующей главе мы начнем писать параллельные программы. Нам потребуется для этого изучить три новых базовых механизма: порождение нового процесса `spawn`, посылка сообщения (с помощью оператора `!`) и получение сообщения `receive`. После чего мы сможем написать несколько простых параллельных программ.

Когда процесс умирает, некоторые другие процессы получают об этом сообщение, если они были с ним связаны. Это все рассматривается в Главе 9 *Ошибки в параллельных программах*.

Когда вы будете читать следующие две главы не забывайте о модели множества людей в одной комнате. Люди это процессы. Люди в комнате имеют каждый свою собственную память - это состояние процесса. Чтобы изменить вашу память, я должен вам что-то сообщить, а вы должны это воспринять и понять. Это и есть посылка и получение сообщений. У нас бывают дети - это порожденные нами процессы. Мы умираем - так заканчивается существование и работа процесса.

Глава 8. Параллельное программирование

В этой главе мы поговорим о процессах. Это маленькие изолированные виртуальные машины, которые могут исполнять функции Эрланга.

Я уверен — вы встречали процессы раньше, но только в контексте операционных систем.

В Эрланге процессы относятся к языку программирования, а НЕ к операционной системе

В Эрланге:

- создание и уничтожение процессов очень быстро;
- посылка сообщений между процессами очень быстрая;
- процессы ведут себя одинаково во всех операционных системах;
- может быть очень большое количество процессов;
- процессы не разделяют память и являются полностью независимыми;
- единственный способ для взаимодействия процессов — это через передачу сообщений.

По этим причинам Эрланг иногда называют *языком с чистой передачей сообщений*.

Если вы раньше не программировали процессы, то до вас доходили слухи о том, что это достаточно трудно. Возможно, вы слышали ужасные истории о нарушениях памяти (memory violations), race conditions, искажении разделяемой памяти (shared-memory corruption) и тому подобном. В Эрланге программировать процессы легко. Для этого нужно только три примитива: `spawn`, `send` и `receive`.

8.1 Параллельные примитивы

Всё, чему мы научились о последовательном программировании, верно и для параллельного. Единственное, что нам надо сделать — это добавить следующие примитивы:

```
Pid = spawn(Fun)
```

Создаёт новый параллельный процесс, который вычисляет (evaluates) `Fun`. Новый процесс работает параллельно с вызвавшим его. `Spawn` возвращает `Pid` (сокращение для *идентификатор процесса*). Вы можете использовать `Pid` для посылки сообщений процессу.

```
Pid ! Message
```

Посыпает сообщение `Message` процессу с идентификатором `Pid`. Посылка сообщения асинхронна. Отправитель не ждёт, а продолжает делать то, чем занимался. `!` называется оператором `send`.

`Pid ! M` определяется как `M` — примитив отправки сообщения `!` возвращает само сообщение. Поэтому `Pid1 ! Pid2 ! ... ! M` означает отправку сообщения `M` всем процессам — `Pid1`, `Pid2` и т. д.

```
receive ... end
```

Принимает сообщение, которое было послано процессу. У него следующий синтаксис:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
end
```

Когда сообщение прибывает к процессу система пытается сопоставить его с образцом `Pattern1` (возможно с учётом условия `Guard1`). Если это выполнилось успешно, то она вычисляет выражение `Expression1`. Если первый образец не совпадает, то она использует `Pattern2` и т.д. Если ни один из образцов не соответствует, сообщение сохраняется для последующей обработки, а процесс ожидает следующего сообщения. Это объясняется подробнее в части 8.6 Избирательный приём на стр. 12.

Образцы и условия в операторе приёма имеют точно такую же синтаксическую форму и значение, как образцы и условия, которые мы используем, когда определяем функцию.

8.2 Простой пример

Помните, как мы писали функцию `area/1` в части 3.1 Модули. Просто, чтобы напомнить вам, код, который определял функцию выглядел вот так:

[Скачать geometry.erl](#)

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R}) -> 3.14159 * R * R.
```

Теперь перепишем эту же функцию как процесс:

[Скачать area_server0.erl](#)

```
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
```

```

{rectangle, Width, Ht} ->
    io:format("*Area of rectangle is ~p~n" ,[Width * Ht]),
    loop();
{circle, R} ->
    io:format("*Area of circle is ~p~n" , [3.14159 * R * R]),
    loop();
Other ->
    io:format("*I don't know what the area of a ~p is ~n" ,[Other])
    loop()
end.

```

Мы можем создать процесс, который вычисляет `loop/0` в shell:

```

1> Pid = spawn(fun area_server0:loop/0).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {circle, 23}.
Area of circle is 1661.90
{circle,23}
4> Pid ! {triangle,2,4,5}.
I don't know what the area of a {triangle,2,4,5} is
{triangle,2,4,5}

```

Что здесь произошло? В строке 1 мы создали новый параллельный процесс.

`spawn(Fun)` создаёт параллельный процесс, который вычисляет `Fun`. Он возвращает `Pid`, который печатается как `<0.36.0>`.

В строке 2 мы посылаем сообщение процессу. Это сообщение совпадает с первым образцом в операторе приёма в `loop/0`.

```

loop() ->
receive
    {rectangle, Width, Ht} ->
        io:format("*Area of rectangle is ~p~n" ,[Width * Ht]),
        loop()
    ...

```

По приёму сообщения, процесс печатает площадь прямоугольника. В конце shell печатает `{rectangle,6,10}`. Это потому, что значением `Pid ! Msg` является `Msg`. Если мы отправляем процессу сообщение, которое он не понимает, он печатает предупреждение. Это выполняется кодом `Other ->...` в операторе приёма `receive`.

8.3 Клиент-сервер - введение

Архитектуры клиент-сервер центральные в Эрланге. По традиции клиент-серверные архитектуры включают сеть, которая отделяет клиента от сервера. Наиболее часто присутствуют несколько экземпляров клиента и один сервер. Слово *сервер* часто вызывает образ некоего достаточно тяжёлого программного обеспечения, работающего на специализированной машине.

В нашем случае предполагается гораздо более легковесный механизм. Клиент и сервер в клиент-серверной архитектуре — это раздельные процессы, и для связи между клиентом и сервером используется обычная передача сообщений Эрланга. Как клиент, так и сервер могут работать на одной и той же машине или на двух разных машинах.

Слова *клиент* и *сервер* ссылаются на роли, которые выполняют эти два процесса. Клиент всегда начинает вычисление отправляя запрос к серверу. Сервер вычисляет ответ и отправляет *ответ* клиенту.

Давайте-ка напишем наше первое клиент-серверное приложение. Начнём вносить небольшие изменения в программу, написанную нами в предыдущей главе.

В предыдущей программе всё, что нам было надо — это послать запрос к процессу, который примет и напечатает этот запрос. Что мы хотим теперь — это послать ответ процессу, который послал первоначальный запрос. Проблема в том, что мы не знаем кому слать ответ. Чтобы сервер послал ответ, клиент должен включить адрес, на который сервер сможет ответить. Это подобно отправке письма кому-то — если вы хотите получить ответ, вам лучше бы указать в письме ваш адрес!

Итак, отправитель должен включить обратный адрес в сообщение. Этого можно достичь, поменяв это:

```
Pid ! {rectangle, 6, 10}
```

на это:

```
Pid ! {self(),{rectangle, 6, 10}}
```

`self()` — это `PID` клиентского процесса.

Для ответа на запрос нам придётся поменять код, принимающий запросы с такого:

```
loop() ->  
    receive
```

```
{rectangle, Width, Ht} ->
    io:format("*Area of rectangle is ~p~n", [Width * Ht]),
    loop()
...

```

на такой:

```
loop() ->
receive
    {From, {rectangle, Width, Ht}} ->
        From ! Width * Ht,
        loop();
...

```

Заметьте, как теперь мы посылаем результат наших вычислений обратно к процессу, определяемому параметром From. Клиент примет результат, т.к. он устанавливает этот параметр в свой собственный идентификатор процесса.

Процесс, который посылает начальный запрос называется *клиентом*. Процесс, который принимает запрос и отправляет ответ называется *сервером*.

В итоге, мы добавили маленькую полезную функцию, названную `rpc` (сокращение для *remote procedure call* — удалённый вызов процедуры), которая включает в себя посылку запроса на сервер и ожидание ответа:

[Скачать area_server1.erl](#)

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
```

Сложив всё это вместе, мы получим следующее:

[Скачать area_server1.erl](#)

```
-module(area_server1).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
```

```

    Response
end.

loop() ->
receive
    {From, {rectangle, Width, Ht}} ->
        From ! Width * Ht,
        loop();
    {From, {circle, R}} ->
        From ! 3.14159 * R * R,
        loop();
    {From, Other} ->
        From ! {error,Other},
        loop()
end.

```

Мы можем поэкспериментировать с этим в шелле:

```

1> Pid = spawn(fun area_server1:loop/0).
<0.36.0>
2> area_server1:rpc(Pid, {rectangle,6,8}).
48
3> area_server1:rpc(Pid, {circle,6}).
113.097
4> area_server1:rpc(Pid, socks).
{error,socks}

```

С этим кодом есть небольшая проблема. В функции `rpc/2` мы посылаем запрос к серверу и ждём ответа. Но мы ждём не ответа от сервера, мы ждём любое сообщение. Если какой-нибудь другой процесс пошлёт клиенту сообщение, в то время как он ждёт ответа от сервера, он (клиент) ошибочно истолкует это сообщение как ответ от сервера. Мы можем исправить это, поменяв вид оператора приёма на такой:

```

loop() ->
receive
    {From, ...} ->
        From ! {self(), ...},
        loop()
    ...

```

и поменяв `rpc` на следующее:

```

rpc(Pid, Request) ->
    Pid ! {self(), Request},

```

```
receive
    {Pid, Response} ->
        Response
end.
```

Как это работает? Когда мы выполняем функцию `rpc`, `Pid` уже связан с каким-то значением, так что в образце `{Pid, Response}` `Pid` привязан к какому-то значению, а `Response` нет. Этот образец совпадёт только с сообщением, состоящим из двухэлементного кортежа, первый элемент которого `Pid`. Все другие сообщения будут поставлены в очередь. (`receive` обеспечивает то, что называется 8.6 Избирательный приём, который я опишу после этой главы).

С этим изменением мы получим следующее:

[Скачать area_server2.erl](#)

```
-module(area_server2).
-export([loop/0, rpc/2]).


rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From ! {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error, Other}},
            loop()
    end.
```

Это работает как и ожидается:

```
1> Pid = spawn(fun area_server2:loop/0).
<0.37.0>
3> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

Есть одно финальное улучшение, которое мы можем сделать. Мы можем скрыть spawn и grpc внутри модуля. Это хорошая практика, т.к. мы сможем менять внутренние детали сервера без изменения кода клиента. В конце мы получаем это:

[Скачать area_server_final.erl](#)

```
-module(area_server_final).
-export([start/0, area/2]).

start() -> spawn(fun loop/0).

area(Pid, What) ->
    rpc(Pid, What).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From ! {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

Для запуска этого мы вызываем функции `start/0` и `area/2` (где раньше мы вызывали `spawn` и `grpc`). Имена лучше те, которые более точно описывают то, что делает сервер:

```
1> Pid = area_server_final:start().
<0.36.0>
2> area_server_final:area(Pid, {rectangle, 10, 8}).
80
4> area_server_final:area(Pid, {circle, 4}).
50.2654
```

8.4 Как долго занимает создать процесс?

В этом месте вы можете начать волноваться о производительности. В конце концов, если мы создаём сотни или тысячи эрланговых процессов, мы должны как-то расплачиваться за это. Давайте поищем — как.

Чтобы исследовать это мы измерим время, нужное для порождения большого количества процессов. Вот программа:

[Скачать processes.erl](#)

```
-module(processes).
-export([max/1]).

%% max(N)
%% Create N processes then destroy them
%% See how much time this takes

max(N) ->
    Max = erlang:system_info(process_limit),
    io:format("*Maximum allowed processes:~p~n*", [Max]),
    statistics(runtime),
    statistics(wall_clock),
    L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N,
    U2 = Time2 * 1000 / N,
    io:format("*Process spawn time=~p (~p) microseconds~n*", [U1, U2]).

wait() ->
    receive
        die -> void
    end.

for(0, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Вот результаты, которые я получил на компьютере, который я использовал для написания этой книги - 2.40GHz Intel Celeron с 512 МБ ОЗУ под управлением Ubuntu Linux:

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time=3.50000 (9.20000) microseconds
ok
2> processes:max(40000).
Maximum allowed processes:32768
=ERROR REPORT==== 26-Nov-2006::14:47:24 ===
Too many processes
...
```

Порождение 20,000 процессов заняло в среднем 3,5 мкс/процесс процессорного времени и 9,2 мкс прошедшего (по часам) времени.

Заметьте, что я использовал встроенную функцию (BIF) `erlang:system_info(process_limit)` для нахождения максимального разрешенного количества процессов. Заметьте, что некоторые из них зарезервированы, так что ваша программа не может на самом деле использовать это количество. Когда мы превышаем системный лимит система рушится с сообщением об ошибке (команда 2).

Системный лимит установлен в 32,767 процессов. Чтобы превысить этот лимит вам придётся запустить эмулятор Эрланга с параметром +P как здесь:

```
$ erl +P 500000
1> processes:max(50000).
Maximum allowed processes:500000
Process spawn time=4.60000 (10.8200) microseconds
ok
2> processes:max(200000).
Maximum allowed processes:500000
Process spawn time=4.10000 (10.2150) microseconds
3> processes:max(300000).
Maximum allowed processes:500000
Process spawn time=4.13333 (73.6533) microseconds
```

В предыдущем примере я установил системный лимит в полмиллиона процессов. Мы можем видеть, что время порождения процесса по существу постоянно между 50,000 и 200,000 процессов. При 300,000 процессов процессорное время порождения остаётся постоянным, но прошедшее время увеличивается в 7 раз. Также я слышу, как вибрирует мой диск. Это верный знак того, что система выполняет подкачку и у меня недостаточно физической памяти для работы с 300,000 процессов.

8.5 Приём с таймаутом

Иногда оператор приёма может вечно ждать сообщения, которое так никогда и не придёт. Для этого может быть несколько причин. Например, может быть логическая ошибка в нашей программе или процесс, который собирался отправить нам сообщение рухнул до отправки.

Чтобы избежать проблем мы можем добавить таймаут в оператор приёма. Он устанавливает максимально время, которое процесс будет ждать при получении сообщения. Синтаксис этого следующий:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
after Time ->
    Expressions
end.
```

Если в течение Time миллисекунд в оператор приёма не придёт ни одного совпадающего сообщения, то процесс перестаёт ждать сообщения и вычисляет Expressions.

Приём только с таймаутом

Вы можете написать `receive`, содержащий только таймаут. Используя это, мы можем определить функцию `sleep(T)`, которая останавливает текущий процесс на время `T` миллисекунд.

[Скачать lib_misc.erl](#)

```
sleep(T) ->
    receive
        after T ->
            true
    end.
```

Приём с нулевым таймаутом

Значение таймаута 0 приводит к немедленному срабатыванию таймаута, но перед тем, как это случится, система пытается сопоставить хоть какой-нибудь образец из почтового ящика. Мы можем использовать это для определения функции `flush_buffer`, которая полностью опустошает почтовый ящик процесса:

[Скачать lib_misc.erl](#)

```
flush_buffer() ->
    receive
        _Any ->
            flush_buffer()
    after 0 ->
        true
    end.
```

Без оператора таймаута функция `flush_buffer` остановилась бы навечно и не вернула бы ничего, если бы почтовый ящик был пуст. Мы также можем использовать нулевой таймаут для создания некоей формы «приоритетного приёма», как в следующем примере:

[Скачать lib_misc.erl](#)

```
priority_receive() ->
    receive
        {alarm, X} ->
            {alarm, X}
    after 0 ->
        receive
            Any ->
                Any
        end
    end.
```

Если в почтовом ящике есть сообщение, не соответствующее образцу `{alarm, X}`, то `priority_receive` примет первое сообщение из почтового ящика. Если же никаких сообщений нет, то приём приостановится на внутреннем операторе `receive` до прихода любого сообщения. Если есть сообщение, соответствующее `{alarm, X}`, то это сообщение будет немедленно возвращено как результат. Помните, что секция `after` проверяется только после проверки на соответствие шаблону всех сообщений из почтового ящика.

Без оператора `after 0` сообщение `alarm` не сработало бы первым.

Замечание: использование больших почтовых ящиков совместно с приоритетным приёмом достаточно неэффективно, так что, если вы собираетесь использовать эту технику, убедитесь, что ваши почтовые ящики не слишком большие.

Приём с бесконечным таймаутом

Если значением таймаута в операторе приёма является атом `infinity`, то таймаут никогда не сработает. Это может быть полезным для программ, в которых значение таймаута вычисляется вне оператора `receive`. Иногда вычисление может захотеть вернуть какое-то конкретное значение, а иногда оно может захотеть, чтобы `receive` ждал вечно.

Организация таймера

Мы можем организовать простой таймер, используя таймауты в приёме.

Функция `stimer:start(Time, Fun)` вычислит `Fun` (функцию без аргументов) после ожидания `Time` миллисекунд. Она возвращает обработчик (который на самом деле `PID`), который может использоваться для отмены таймера при необходимости.

[Скачать stimer.erl](#)

```
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).

cancel(Pid) -> Pid ! cancel.

timer(Time, Fun) ->
    receive
        cancel ->
            void
        after Time ->
            Fun()
    end.
```

Мы можем проверить это следующим образом:

```
1> Pid = stimer:start(5000, fun() -> io:format("timer event~n") end).
<0.42.0>
timer event
```

Здесь я ждал больше пяти секунд, чтобы сработал таймер. Сейчас я запущу таймер и отменю его до того, как выйдет таймерное время:

```
2> Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

8.6 Избирательный приём

До сих пор мы проходили по верхушкам того, как действительно работают `send` и `receive`. `Send` на самом деле не отправляет сообщения процессу. В место того, `send` отправляет сообщение в почтовый ящик процесса, а `receive` пытается удалить сообщения из почтового ящика.

Каждый процесс в Эрланге имеет свой собственный почтовый ящик. Когда вы посыпаете сообщение процессу, это сообщение помещается в почтовый ящик. Почтовый ящик проверяется только тогда, когда программа вычисляет оператор `receive`:

```
receive
  Pattern1 [when Guard1] ->
    Expressions1;
  Pattern2 [when Guard1] ->
    Expressions1;
  ...
  after Time ->
    ExpressionTimeout
end.
```

`receive` работает следующим образом:

1. Когда мы входим в оператор `receive`, мы запускаем таймер (но только, если в выражении присутствует секция `after`).
2. Взять первое сообщение из почтового ящика и попытаться соотнести его с образцами `Pattern1`, `Pattern2` и т.д. Если соответствие успешно, то сообщение удаляется из почтового ящика и вычисляется выражение, следующее за образцом.
3. Если ни один из образцов в операторе `receive` не соответствует первому сообщению из почтового ящика, то первое сообщение удаляется из ящика и помещается в «отложенную очередь» (`save queue`). Затем так же проверяется второе сообщение. Эта процедура повторяется до тех пор, пока не будет найдено совпадающее сообщение, либо не будут проверены все сообщения из почтового ящика.
4. Если ни одно сообщение из почтового ящика не соответствует, процесс приостанавливается и ждёт до тех пор, пока новое сообщение не будет помещено в почтовый ящик. Заметьте, что когда новое сообщение прибывает, сообщения из отложенной очереди не проверяются заново на соответствие образцам. Проверяется только новое сообщение.
5. Как только сообщение совпало с образцом, сразу после этого все сообщения из отложенной очереди помещаются обратно в почтовый ящик в том же порядке, в

каком они прибыли к процессу. Если был установлен таймер, то он очищается.

6. Если таймер истёк, пока мы ждали сообщение, то выполнится выражение `ExpressionTimeout`, после чего все отложенные сообщения поместятся обратно в почтовый ящик в том же порядке, в каком они прибыли к процессу.

8.7 Зарегистрированные процессы

Если мы хотим послать сообщение процессу, нам надо знать его `PID`. Это часто не удобно, т.к. `PID` надо передать всем процессам в системе, желающим взаимодействовать с данным процессом. С другой стороны, это очень безопасно. Если вы не раскрываете `PID` процесса, другие процессы не могут взаимодействовать с ним никаким образом.

У Эрланга есть метод *публикации* идентификатора процесса, так что любой процесс в системе может общаться с этим процессом. Такой процесс называется *зарегистрированным процессом*. Есть четыре встроенные функции (BIF) для управления зарегистрированными процессами:

```
register(AnAtom, Pid)
```

зарегистрировать процесс `Pid` с именем `AnAtom`. Регистрация не успешна, если `AnAtom` уже был использован для регистрации процесса.

```
unregister(AnAtom)
```

удалить любые регистраций, связанные с `AnAtom`.

Замечание: если зарегистрированный процесс умирает, он автоматически разрегистрируется

```
whereis(AnAtom) -> Pid | undefined
```

находит, где зарегистрирован `AnAtom`. Возвращает идентификатор процесса `Pid`, либо возвращает атом `undefined`, если никакой процесс не связан с `AnAtom`.

```
registered() -> [AnAtom::atom()]
```

возвращает список зарегистрированных процессов в системе.

Используя `register`, мы можем пересмотреть пример из части 8.2 Простой пример на стр. 2 и можем попытаться зарегистрировать имя процесса, который мы создали:

```
1> Pid = spawn(fun area_server0:loop/0).  
<0.51.0>  
2> register(area, Pid).  
true
```

Как только имя зарегистрировано, мы можем отправить ему сообщение подобно этому:

```
3> area ! {rectangle, 4, 5}.  
Area of rectangle is 20  
{rectangle,4,5}
```

Часы

Мы можем использовать регистрацию при создании процесса, который представляет из себя часы:

[Скачать clock.erl](#)

```
-module(clock).  
-export([start/2, stop/0]).  
  
start(Time, Fun) ->  
    register(clock, spawn(fun() -> tick(Time, Fun) end)).  
  
stop() -> clock ! stop.  
  
tick(Time, Fun) ->  
    receive  
        stop ->  
            void  
        after Time ->  
            Fun(),  
            tick(Time, Fun)  
    end.
```

Часы будут радостно отстукивать, пока вы не остановите их:

```
3> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).  
true  
  
TICK {1164,553538,392266}  
  
TICK {1164,553543,393084}
```

```
TICK {1164,553548,394083}
```

```
TICK {1164,553553,395064}
```

```
4> clock:stop().
```

```
stop
```

8.8 Как нам писать параллельную программу?

Когда я пишу параллельную программу, то почти всегда я начинаю с чего-то подобного:

[Скачать ctemplate.erl](#)

```
-module(ctemplate).
-compile(export_all).

start() ->
    spawn(fun() -> loop([]) end).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop(X) ->
    receive
        Any ->
            io:format("Received:~p~n" ,[Any]),
            loop(X)
    end.
```

Цикл приёма — это просто пустой цикл, который принимает и печатает все сообщения, которые я посылаю ему. По мере разработки программы я начинаю посыпать сообщения процессу. Т.к. я начинаю цикл приёма вообще без образцов, которые соответствуют сообщениям, то получу распечатку из кода в конце оператора приёма. Когда это происходит, я добавляю образец соответствия в цикл приёма и перезапускаю программу. Эта техника в значительной степени определяет порядок, в котором я пишу программы — я начинаю с небольшой программы, постепенно

увеличиваю её, тестируя по мере написания.

8.9 Слово о хвостовой рекурсии

Взгляните на цикл приёма в сервере вычисления площади, который мы писали ранее:

[Скачать area_server_final.erl](#)

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From ! {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

Если вы посмотрите внимательно, то увидите, что каждый раз, когда мы принимаем сообщение, мы обрабатываем это сообщение и затем сразу же снова вызываем `loop()`. Такая процедура называется *хвостовой рекурсией*. Функция с хвостовой рекурсией может быть скомпилирована так, что последний вызов функции в последовательности операторов может заменяться обычным переходом на начало функции, которую вызывали. Это значит, что функция с хвостовой рекурсией может зацикливаться бесконечно без потребления стека.

Допустим, мы написали следующий (неправильный) код:

```
loop() ->
    {From, {rectangle, Width, Ht}} ->
        From ! {self(), Width * Ht},
        loop(),
        someOtherFunc();
    {From, {circle, R}} ->
        From ! {self(), 3.14159 * R * R},
        loop();
    ...
end.
```

В строке 4 мы вызываем `loop()`, но компьютер должен сообразить что «после вызова

`loop()`, мне придётся вернуться сюда, так как надо будет вызвать функцию `someOtherFunc()` в строке 5». Поэтому он сохраняет адрес `someOtherFunc` в стеке и переходит к началу `loop()`. Проблема тут в том, что `loop()` никогда не возвращается. Вместо этого она зацикливается навечно. Так что каждый раз, когда мы проходим строку 4 адрес возврата заносится в стек. И в конце концов у системы заканчивается место.

Избежать этого легко — если вы пишите функцию `F`, которая никогда не возвращается (такую как `loop()`), убедитесь, что вы никогда ничего не вызываете после вызова `F` и не используйте `F` в создании кортежей или списков.

8.10 Порождение с MFA

Большинство программ, которые мы пишем используют `spawn(Fun)` для создания нового процесса. Это прекрасно до тех пор, пока мы не захотим обновлять наш код на ходу. Иногда мы хотим написать код, который можно обновлять в то время, как он выполняется. Если мы хотим быть уверенными в том, что наш код может обновляться динамически, то нам надо использовать другую форму `spawn`.

```
spawn(Mod, FuncName, Args)
```

это создаёт новый процесс. `Args` — это список аргументов вида `[Arg1, Arg2, ..., ArgN]`. Вновь созданный процесс начинает вычисление `Mod:FuncName(Arg1, Arg2, ..., ArgN)`.

Порождение функции с явным указанием модуля, имени функции и списка аргументов (называемые MFA) — это верный способ быть уверенными в том, что наши процессы будут корректно обновляться новыми версиями кода модуля, если он компилируется и в то же время используется. Механизм динамического обновления кода не работает с порождёнными функциями. Он работает только с явно указанными MFA. За дальнейшими деталями читайте приложение Е.4 *Динамическая загрузка кода*.

8.11 Проблемы

Напишите функцию `start(AnAtom, Fun)`, чтобы зарегистрировать `AnAtom` как `spawn(Fun)`. Убедитесь, что ваша программа работает корректно в случае, когда два параллельных процесса одновременно вычисляют `start/2`. В этом случае вы должны гарантировать, что один из этих процессов преуспеет, а другой потерпит неудачу.

Напишите кольцевой тест. Создайте `N` процессов в кольце. Отправьте сообщение по

кольцу M раз так, чтобы было отправлено $N * M$ сообщений. Замерьте время, которое тратится для разных значений N и M .

Напишите подобную программу на каком-нибудь другом языке программирования, известном вам. Сравните результаты. Напишите блог и опубликуйте результаты в Интернете!

Вот и всё — теперь вы можете писать параллельные программы!

Дальше мы рассмотрим восстановление после ошибок и увидим, как мы можем писать параллельные программы, устойчивые к сбоям, используя три дополнительные концепции: линки, сигналы и перехват завершения процессов. Это в следующей главе.

Глава 9. Ошибки в параллельных программах

Ранее мы видели как ловить ошибки в последовательных программах. В этой главе мы расширим механизм обработки ошибок на параллельные программы.

Это второй и последний этап в понимании того, как Эрланг обрабатывает ошибки. Чтобы понять это нам надо ввести три новые концепции: *связи* (*links*), *сигналы выхода* (*exit signals*) и идею *системного процесса* (*system process*).

9.1 Связанные процессы

Если один процесс как-то зависит от другого, то он, вполне возможно, захочет присмотреть за его здоровьем. Один из способов это сделать - посредством встроенной функции Эрланга `link`. (Другой способ сделать это — используя мониторинг, который описан в руководстве по эрлангу).

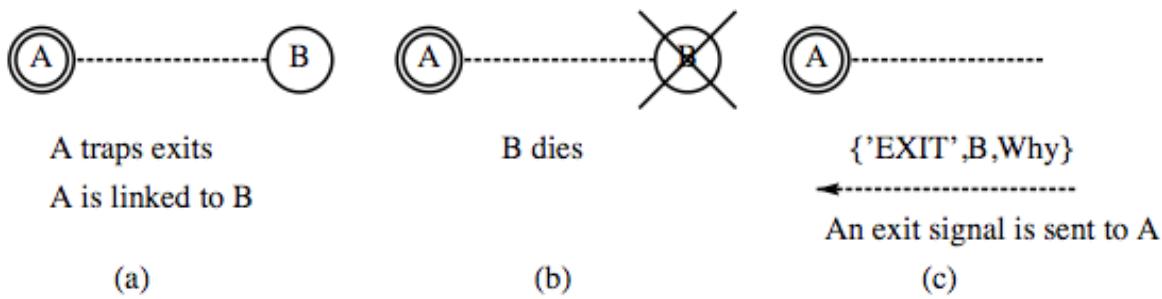


Рисунок 9.1: Сигналы выхода и связи

На Рис. 9.1 показаны процессы A и B . Они связаны между собой (показано прерывистой линией). Связь была установлена, когда один из процессов вызвал

встроенную функцию `link(P)` с параметром `P` — идентификатором другого процесса. После установления связи оба процесса неявно следят друг за другом. Если умрёт `A`, то `B` получит сигнал выхода (`exit signal`). И наоборот — если умрёт `B`, то такой сигнал получит `A`.

Механизмы, описанные в этой главе, совершенно общие. Они работают на одном-единственном узле системы и на нескольких узлах в распределенной эрланговой системе. Как мы увидим в Главе 10 «Распределённое программирование» мы можем порождать процессы на удалённых узлах так же легко, как и на текущем узле. Все механизмы для связи, которые мы обсуждаем в этой главе, работают так же хорошо и в распределённой системе.

Что происходит, когда процесс получает сигнал выхода? Если не принять специальных мер, то процесс, получивший сигнал выхода, завершится. Однако процесс может перехватывать такие сигналы выхода. В этом случае он называется *системным процессом* (*system process*). Если процесс, связанный с системным процессом, завершается по каким-либо причинам, то системный процесс не завершается автоматически. Вместо этого он получает сигнал выхода, который можно перехватить и обработать.

Часть (а) на Рис. 9.1 показывает связанные процессы. `A` — это системный процесс (показан в двойном кольце). В части (б) `B` умирает, и в части (с) сигнал выхода посыпается к `A`.

Позже в этой части мы рассмотрим со всеми подробностями то, что происходит, когда процессу приходит полный сигнал выхода. Но до этого мы начнём с небольшого примера, который покажет — как использовать этот механизм для написания простого обработчика выхода. Обработчик выхода — это процесс, который исполняет заданную функцию, когда какой-либо другой процесс завершается аварийно. Обработчик выхода — это полезный строительный блок для создания более развитых абстракций.

9.2 Обработчик `on_exit`

Мы хотим выполнить некое действие, когда процесс завершается. Можно написать функцию `on_exit(Pid, Fun)`, которая устанавливает связь с процессом `Pid`. Если `Pid` умирает с причиной `Why`, то вычисляется функция `Fun(Why)`.

Вот эта программа:

[Скачать lib_misc.erl](#)

```
on_exit(Pid, Fun) ->
```

```

spawn(fun() ->
    process_flag(trap_exit, true),
    link(Pid),
    receive
        {'EXIT', Pid, Why} ->
            Fun(Why)
    end
end).

```

В строке 3 выражение `process_flag(trap_exit, true)` превращает порождённый процесс в системный. `link(Pid)` в строке 4 связывает вновь созданный процесс с `Pid`. В конце, когда процесс умирает, принимается (строка 6) и обрабатывается (строка 7) сигнал выхода.

Замечание: когда вы прочитаете код этой программы, вы увидите, что мы используем `Pid` повсеместно. Это `Pid` связанного процесса. Нежелательно использовать имя переменной вроде `LinkedPid`, т.к. до вызова `link(Pid)` связь с этим процессом ещё не установлена. Когда вы видите сообщение типа такого `{'EXIT', Pid, _}` вы должны понять, что `Pid` — это связанный процесс и он только что умер.

Чтобы проверить всё это, определим функцию `F`, которая ждёт единственное сообщение `X` и затем вычисляет `list_to_atom(X)`:

```

1> F = fun() ->
receive
X -> list_to_atom(X)
end
end.

#Fun<erl_eval.20.69967518>

```

Создадим процесс:

```

2> Pid = spawn(F).
<0.61.0>

```

И установим обработчик `on_exit` для мониторинга этого процесса:

```

3> lib_misc:on_exit(Pid,
fun(Why) ->
io:format(" ~p died with:~p~n",[Pid, Why])
end).
<0.63.0>

```

Если теперь мы отправим атом к `Pid`, то процесс `Pid` умрёт (т.к. он попытается выполнить `list_to_atom`, а входные данные у него — не список) и тогда вызовется обработчик `on_exit`:

```
4> Pid ! hello.  
hello  
<0.61.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}
```

Функция, которая вызывается при умирании процесса может, разумеется, делать всё, что ей угодно — она может проигнорировать ошибку, зарегистрировать (log) ошибку или перезапустить приложение. Выбор зависит от программиста.

9.3 Дистанционная обработка ошибок

Давайте остановимся и немного подумаем над предыдущим примером. Он показывает исключительно важную часть философии Эрланга, называемую *дистанционной обработкой ошибок*.

Поскольку эрланговая система состоит из множества параллельных процессов, то нам не приходится иметь дело с ошибками прямо в том процессе, где они происходят — мы можем работать с ними в отдельном процессе. Процесс, который имеет дело с ошибкой даже *не обязан находиться на той же самой машине*. В распределённом Эрланге, описанном в следующей главе, мы увидим, что этот простой механизм работает даже между разными машинами. Это очень важно, т.к. если вся машина выходит из строя, то программа, которая исправляет такую ошибку должна находиться на какой-нибудь другой машине.

9.4 Детали обработки ошибок

Давайте снова глянем на те три концепции, которые лежат в основе эрланговой обработки ошибок:

Связи

Связь — это нечто, что определяет путь распространения ошибок между двумя процессами. Если два процесса связаны вместе и один из них умирает, то другому процессу посыпается *сигнал выхода*.

Сигналы выхода

Сигнал выхода — это нечто, что создаётся процессом, когда тот умирает. Этот сигнал

рассылается всем процессам, которые связаны с умершим. Сигнал выхода содержит информацию о том, почему умер процесс. Причина может быть любым элементом данных Эрланга. Причина может быть явно указана посредством вызова `exit(Reason)` или неявно при возникновении ошибки. Например, если программа выполняет деление числа на ноль, то причина ошибки устанавливается в атом `badarith`.

Когда процесс завершает выполнять функцию, с которой он был вызван, причина выхода устанавливается в `normal`.

Дополнительно процесс `Pid1` может послать сигнал выхода `X` процессу `Pid2`, выполнив функцию `exit(Pid2, X)`. Процесс, который посыпает сигнал выхода не умирает. Он продолжает выполнение после отправки сигнала. `Pid2` получит сообщение `{'EXIT', Pid1, X}` (это если он перехватывает сигналы выхода, т.е. является системным процессом) в точности, как если бы исходный процесс умер. Используя этот механизм, процесс `Pid1` может «подделать» собственную смерть (умышленно).

Системные процессы

Когда процесс получает ненормальный сигнал выхода, он тоже завершается, если только это не специальный процесс, называемый *системным процессом*. Когда системный процесс получает сигнал выхода `Why` от процесса `Pid`, этот сигнал преобразуется в сообщение `{'EXIT', Pid, Why}` и добавляется в почтовый ящик системного процесса.

Вызов встроенной функции `process_flag(trap_exit, true)` превращает обычный процесс в системный, который может перехватывать сигналы выхода.

Когда процесс получает *сигнал выхода* может произойти несколько вещей. Что именно произойдёт, зависит от состояния процесса и значения сигнала выхода и определяется следующей таблицей:

<code>trap_exit</code>	<code>Сигнал выхода</code>	<code>Действие</code>
<code>true</code>	<code>kill</code>	Умереть: рассылка сигнала выхода <code>killed</code> по всем связям
<code>true</code>	<code>X</code>	Добавить <code>{'EXIT', Pid, X}</code> в почтовый ящик
<code>false</code>	<code>normal</code>	Продолжить: ничего не делающий сигнал удаляется
<code>false</code>	<code>kill</code>	Умереть: рассылка сигнала выхода <code>killed</code> по всем связям

false

X

Умереть: рассылка сигнала выхода X по всем связям

Если указана причина `kill`, то рассыпается *неперехватываемый сигнал выхода*. Такой сигнал всегда убивает процесс, даже если это системный процесс. Это используется в OTP процессом-супервизором для принудительного завершения сбойных процессов. Когда процесс получает сигнал `kill`, он умирает и сигналы `killed` рассыпаются по всем его связям. Это является мерой предосторожности, чтобы случайно не убить больше от системы, чем необходимо.

Сигнал `kill` предназначен для убийства сбойных процессов. Хорошенько подумайте прежде чем использовать его.

Особенности программирования перехвата выхода

Перехват выхода обычно гораздо легче, чем вы могли бы подумать, прочитав предыдущие главы. И хотя механизмы выхода и его перехвата можно использовать рядом хитроумных способов, большинство программ используют три простых подхода.

Подход 1: Меня не волнует, если процесс, который я создал, падает

Процесс создаёт другой процесс, используя функцию `spawn`:

```
Pid = spawn(fun() -> ... end)
```

Ничего более. Если порождённый процесс падает, то текущий процесс продолжает работать.

Подход 2: Я хочу умереть, если процесс, который я создал, падает**

Если быть точным, мы должны бы сказать «Если процесс, который я создал, падает по причине, отличной от `normal`». Чтобы достичь этого исходный процесс использует функцию `spawn_link` и не должен заранее готовиться к перехвату выхода. Можно просто написать так:

```
Pid = spawn_link(fun() -> ... end)
```

Теперь, если порождённый процесс падает по причине, отличной от `normal`, текущий процесс также падает.

Подход 3: Я хочу обработать ошибки, если процесс, который я создал, падает

Здесь мы используем `spawn_link` и `trap_exit`. Код будет таким:

```
...
```

```

process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
...
loop(...).

loop(State) ->
    receive
        {'EXIT', SomePid, Reason} ->
            %% do something with the error
            loop(State1);
        ...
    end.

```

Теперь процесс, вычисляющий `loop`, перехватывает выход и не умрёт, если упадёт связанный с ним другой процесс. Он увидит все сигналы выхода (преобразованные в сообщения) от умирающего процесса и сможет предпринять все необходимые действия, когда обнаружит сбой.

Перехват сигналов выхода (дальнейшее развитие)

Вы можете пропустить эту часть, если читаете в первый раз. Большинство из того, что вы захотите сделать, может быть обработано тремя подходами из предыдущей части. Если же вы хотите знать, как это работает на самом деле — читайте далее. Но учите — вас предупреждали. Детали этих механизмов возможно будет трудно понять. В большинстве случаев вам не понадобится вникать в этот механизм, особенно если вы используете один из общих подходов (из предыдущей части) или библиотеки OTP — система всё сделает за вас правильным образом.

Для действительного понимания подробностей обработки ошибок мы напишем небольшую программу. Она покажет как взаимодействуют обработка ошибок и связи. Программа начинается вот так:

[Скачать edemo1.erl](#)

```

-module(edemo1).
-export([start/2]).

start(Bool, M) ->
    A = spawn(fun() -> a() end),
    B = spawn(fun() -> b(A, Bool) end),
    C = spawn(fun() -> c(B, M) end),
    sleep(1000),
    status(b, B),
    status(c, C).

```

Она запускает три процесса: A, B, C. Мысль в том, что A будет связан с B, а B будет связан с C. A будет перехватывать выход и наблюдать за выходом B. B будет перехватывать выход, если Bool будет true. А C умрёт с причиной M.

(Вы, возможно, удивитесь по поводу sleep(1000). Это для того, чтобы сообщения, приходящие при смерти C, вывелись перед проверкой состояния процессов. Это не меняет логику программы, но влияет на порядок вывода.)

Здесь код для всех трёх процессов:

[Скачать edemo1.erl](#)

```
a() ->
    process_flag(trap_exit, true),
    wait(a).

b(A, Bool) ->
    process_flag(trap_exit, Bool),
    link(A),
    wait(b).

c(B, M) ->
    link(B),
    case M of
        {die, Reason} ->
            exit(Reason);
        {divide, N} ->
            1/N,
            wait(c);
        normal ->
            true
    end.
```

wait/1 всего лишь печатает сообщение, которое принимает:

[Скачать edemo1.erl](#)

```
wait(Prog) ->
receive
    Any ->
        io:format("Process ~p received ~p~n" ,[Prog, Any]),
        wait(Prog)
end.
```

Остаток программы такой:

[Скачать edemo1.erl](#)

```
sleep(T) ->
    receive
        after T -> true
    end.

status(Name, Pid) ->
    case erlang:is_process_alive(Pid) of
        true ->
            io:format("process ~p (~p) is alive~n" , [Name, Pid]);
        false ->
            io:format("process ~p (~p) is dead~n" , [Name,Pid])
    end.
```

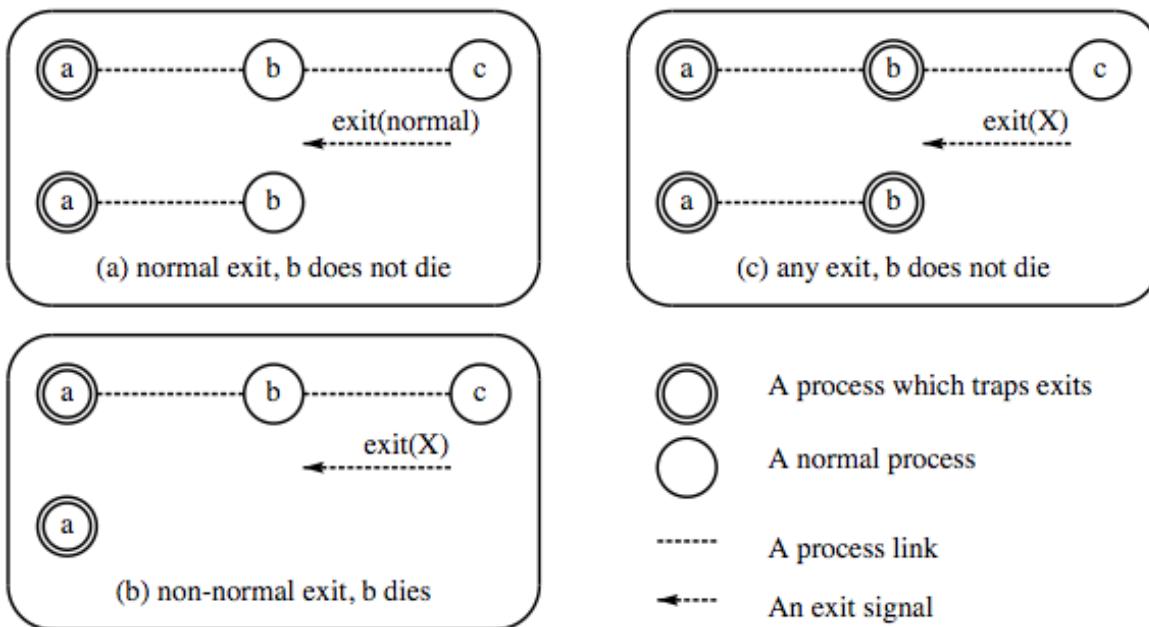


Рисунок 9.2: Перехват сигналов выхода

Теперь запустим программу и будем генерировать различные сигналы выхода в **C** и наблюдать, как это скажется на **B**. При запуске программы смотрите на Рис. 9.2. На нём показано что происходит, когда сигнал выхода приходит от **C**, какие процессы существуют, какие есть связи между ними. Диаграммы состоят из двух частей — часть «до» (верхняя часть) показывает состояние процессов до прихода сигнала выхода и часть «после» (нижняя часть) показывает процессы после того, как средний процесс получил сигнал выхода.

Сначала предположим, что **B** — это обычный процесс (т.е. который не делал `process_flag(trap_exit, true)`):

```
1> edemo1:start(false, {die, abc}).  
Process a received {'EXIT',<0.44.0>,abc}  
process b (<0.44.0>) is dead  
process c (<0.45.0>) is dead  
ok
```

Когда **C** выполняет `exit(abc)` процесс **B** умирает (потому что он не перехватывает выход). При выходе **B** рассыпает полученный сигнал выхода по всем процессам, с которыми он связан. **A** (который перехватывает выход) получает сигнал выхода и превращает его в сообщение об ошибке `{'EXIT', <0.44.0>, abc}`. (Заметьте, что процесс `<0.44.0>` — это процесс **B**, который умирает).

Теперь попробуем другой сценарий. Мы скажем процессу **C** умереть с причиной `normal`.

```
2> edemo1:start(false, {die, normal}).  
process b (<0.48.0>) is alive  
process c (<0.49.0>) is dead  
ok
```

Процесс **B** не умирает, т.к. он получает сигнал выхода `normal`.

Теперь пусть **C** выполнит арифметическую ошибку:

```
3> edemo1:start(false, {divide,0}).  
=ERROR REPORT==== 8-Dec-2006::11:12:47 ===  
Error in process <0.53.0> with exit value: {badarith,[{edemo1,c,2}]}  
Process a received {'EXIT',<0.52.0>,{badarith,[{edemo1,c,2}]}}  
process b (<0.52.0>) is dead  
process c (<0.53.0>) is dead  
ok
```

Когда **C** пытается делить на ноль происходит ошибка и процесс умирает с ошибкой `{badarith, ...}`. Процесс **B** принимает ошибку и тоже умирает, так что ошибка доходит до **A**.

В конце мы заставляем **C** завершиться по причине `kill`:

```
4> edemo1:start(false, {die,kill}).  
Process a received {'EXIT',<0.56.0>,killed} <-- ** замена killed **  
process b (<0.56.0>) is dead  
process c (<0.57.0>) is dead  
ok
```

Причина выхода `kill` заставляет `B` умереть и ошибка распространяется по всем связям `B` как `killed`. Поведение в этих случаях показано на рисунке в частях (а) и (б).

Мы можем повторить эти тесты для случая, когда `B` перехватывает выход. Эта ситуация показана на рисунке в части (с).

```
5> edemo1:start(true, {die, abc}).  
Process b received {'EXIT',<0.61.0>,abc}  
process b (<0.60.0>) is alive  
process c (<0.61.0>) is dead  
ok  
6> edemo1:start(true, {die, normal}).  
Process b received {'EXIT',<0.65.0>,normal}  
process b (<0.64.0>) is alive  
process c (<0.65.0>) is dead  
ok  
7> edemo1:start(true, normal).  
Process b received {'EXIT',<0.69.0>,normal}  
process b (<0.68.0>) is alive  
process c (<0.69.0>) is dead  
8> edemo1:start(true, {die,kill}).  
Process b received {'EXIT',<0.73.0>,kill}  
process b (<0.72.0>) is alive  
process c (<0.73.0>) is dead  
ok
```

Во всех этих случаях `B` перехватывает ошибку. Процесс `B` работает как некий фильтр, ловя все ошибки от `C` и не допуская их к `A`. Мы можем проверить `exit/2` с `code/edemo2.erl`. Эта программа похожа на `edemo1`, отличие только в функции `c/2`, которая вызывает `exit/2`. Вот, как это выглядит:

[Скачать `edemo2.erl`](#)

```
c(B, M) ->  
    process_flag(trap_exit, true),  
    link(B),  
    exit(B, M),  
    wait(c).
```

Запустив `edemo2`, мы увидим следующее:

```
1> edemo2:start(false, abc).  
Process c received {'EXIT',<0.81.0>,abc}
```

```
Process a received {'EXIT',<0.81.0>,abc}
process b (<0.81.0>) is dead
process c (<0.82.0>) is alive
ok
2> edemo2:start(false, normal).
process b (<0.85.0>) is alive
process c (<0.86.0>) is alive
ok
3> edemo2:start(false, kill).
Process c received {'EXIT',<0.97.0>,killed}
Process a received {'EXIT',<0.97.0>,killed}
process b (<0.97.0>) is dead
process c (<0.98.0>) is alive
ok
4> edemo2:start(true, abc).
Process b received {'EXIT',<0.102.0>,abc}
process b (<0.101.0>) is alive
process c (<0.102.0>) is alive
ok
5> edemo2:start(true, normal).
Process b received {'EXIT',<0.106.0>,normal}
process b (<0.105.0>) is alive
process c (<0.106.0>) is alive
ok
6> edemo2:start(true, kill).
Process c received {'EXIT',<0.109.0>,killed}
Process a received {'EXIT',<0.109.0>,killed}
process b (<0.109.0>) is dead
process c (<0.110.0>) is alive
ok
```

9.5 Примитивы для обработки ошибок

Вот наиболее распространённые примитивы для управления связями и для перехвата и отправки сигналов выхода:

```
@spec spawn_link(Fun) -> Pid
```

Это в точности как `spawn(Fun)`, но дополнительно создаёт связь между процессами родителя и потомка. (`spawn_link` — это атомарная операция. Она не эквивалентна последовательным вызовам `spawn` и `link`, т.к. в промежутке между этими двумя вызовами процесс может умереть)

```
@spec process_flag(trap_exit, true)
```

Превращает текущий процесс в системный процесс. Системный процесс — это процесс, который может принимать и обрабатывать сигналы об ошибках.

Замечание: признак `trap_exit` можно установить в `false` после того, как он был установлен в `true`. Этот примитив должен использоваться только для превращения обычного процесса в системный и больше ни для каких других целей.

```
@spec link(Pid) -> true
```

Связывает текущий процесс с процессом `Pid`, если такой связи ещё нет. Связь симметрична. Если процесс `A` выполняет `link(B)`, то он связывается с `B`. Итог этого такой же, как если бы `B` выполнил `link(A)`.

Если процесс `Pid` не существует, то возникает исключение с выходом (exit exception) `noproc`.

Если процесс `A` уже связан с `B` (или наоборот), то вызов игнорируется.

```
@spec unlink(Pid) -> true
```

Удаляет любую связь между текущим процессом и процессом `Pid`.

```
@spec exit(Why) -> none()
```

Завершает текущий процесс с причиной `Why`. Если выполнение `exit` происходит вне пределов

Джо спрашивает...

Как мы можем сделать систему устойчивую к сбоям?

Чтобы сделать что-то устойчивым к сбоям, нам надо, как минимум, два компьютера. Один компьютер будет делать работу, а второй смотреть за первым и быть готовым продолжить работу с момента, когда первый компьютер выйдет из строя.

Именно так и работает восстановление после ошибок в Эрланге. Один процесс делает дело, а второй наблюдает за первым и подхватывает работу, если что-то идёт неправильно. Вот поэтому нам надо мониторить процессы и знать почему что-то идёт неправильно. Примеры этой главы показывают как сделать это.

В распределённом Эрланге процессы, которые делают работу и процессы, которые

наблюдают за теми, которые делают работу, могут быть вообще на разных машинах. Используя такую технику, мы можем создавать программы, устойчивые к сбоям.

Это общий шаблон. Мы называем это моделью *рабочего-наблюдателя* (*worker-supervisor*) и целая секция в библиотеках OTP посвящена построению деревьев наблюдения (*supervision trees*), которые используют эту идею.

Базовый примитив языка, который делает это возможным — это `link`.

Как только вы поймёте как работает `link` и организуете себе доступ к двум компьютерам, вы сможете создать вашу первую устойчивую к сбоям систему.

`catch`, то текущий процесс рассыпает сигнал выхода с причиной *Why* всем процессам, с которыми он связан на текущий момент.

```
@spec exit(Pid, Why) -> true
```

Посыпает сигнал выхода с причиной *Why* к процессу *Pid*.

```
@spec erlang:monitor(process, Item) -> MonitorRef
```

Устанавливает монитор. `Item` — это `PID` или зарегистрированное имя процесса. За подробностями обращайтесь к руководству по Эрлангу.

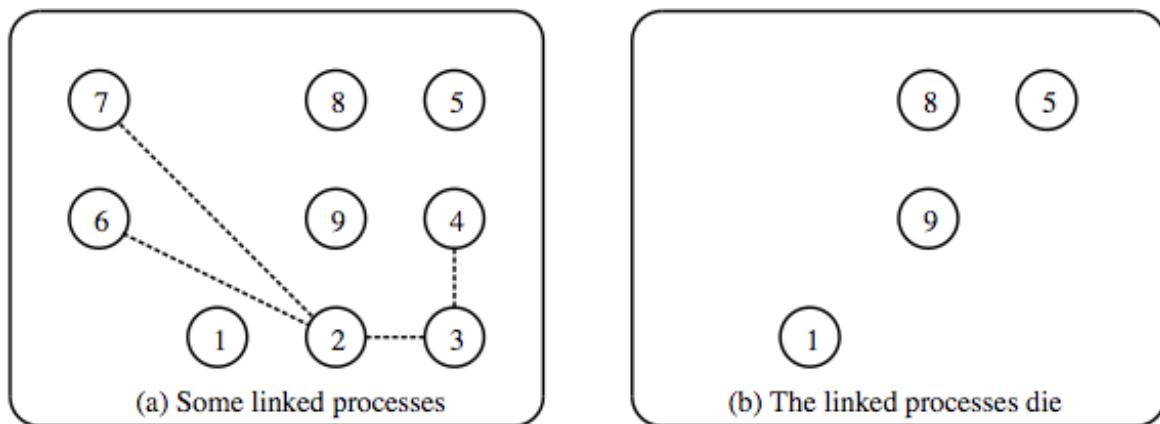


Рисунок 9.3: Перехват сигналов выхода

9.6 Набор связанных процессов

Допустим, у нас есть большой набор параллельных процессов, которые что-то вычисляют и что-то пошло не так. Как найти и убить процессы, которые надо?

Самый лёгкий способ — это убедиться, что все эти процессы связаны и не перехватывают выход. Если какой-либо процесс умирает с причиной, отличной от `normal`, то умирают и все процессы в группе.

Это поведение показано на Рис. 9.3. В части (а) показан набор из девяти процессов, причём процессы 2, 3, 4, 6 и 7 связаны вместе. Если любой из этих процессов умрёт с ненормальной причиной, то умрёт и вся группа процессов, как показано в части (б).

Наборы связанных процессов используются для структурирования программ при создании устойчивых к сбоям систем. Вы можете сделать это сами, либо вы можете воспользоваться библиотечными функциями, описанными в Главе 18.5 *Дерево наблюдения (Supervision Tree)*.

9.7 Мониторы

Программирование связей иногда бывает коварным, т.к. связи являются **дву направленными**. Если `A` умирает, то к `B` будет послан сигнал выхода и наоборот. Чтобы не дать процессу умереть нам приходится делать его системным. Иногда мы не хотим этого делать. В таких случаях мы можем использовать монитор.

Монитор — это односторонняя связь. Если процесс `A` мониторит процесс `B`, и процесс `B` умирает, то к `A` будет послан сигнал выхода. Однако, если `A` умирает, то к `B` не будет послано никакого сигнала выхода. Полное описание возможностей монитора можно найти в руководстве по Эрлангу.

9.8 Постоянный (keep-alive) процесс

Чтобы подвести итог этой главе, создадим постоянный процесс. Идея в том, чтобы создать зарегистрированный процесс, который будет жив всегда — если он по какой-либо причине умирает, то тут же перезапускается.

Мы можем использовать `on_exit`, чтобы достичь этого:

[Скачать lib_misc.erl](#)

```
keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
```

Здесь создаётся зарегистрированный под именем `Name` процесс, который вычисляет `spawn(Fun)`. Если процесс по какой-то причине умирает, то он сразу запускается

заново.

В `on_exit` и `keep_alive` есть достаточно тонкая ошибка. Хотелось бы знать — заметили ли вы её? Когда мы делаем что-то вроде такого:

```
Pid = register(...),  
on_exit(Pid, fun(X) -> ..),
```

есть возможность, что процесс умрёт в промежутке между этими двумя вызовами. Если процесс умирает перед тем, как выполнится `on_exit`, то связь не будет создана и `on_exit` не сработает так, как ожидается. Это может произойти в том случае, если две программы пытаются выполнить `keep_alive` одновременно с одним и тем же значением `Name`. Это называется *race conditions*. Два кусочка кода — этот и часть, которая устанавливает связь внутри `on_exit`, пытаются обогнать друг друга. Если здесь что-нибудь пойдёт не так, то ваша программа может повести себя непредсказуемо.

Я не буду решать эту проблему здесь — подумайте над этим сами. Когда вы объединяете примитивы `spawn`, `spawn_link`, `register` и т.п., вы должны хорошоенько подумать о возможных *race conditions*. Пишите ваш код так, чтобы *race conditions* никогда не возникали.

К счастью, в библиотеках OTP есть готовый код для построения серверов, деревьев наблюдения и т.п. Эти библиотеки хорошо протестированы и не должны содержать никаких *race conditions*. Используйте эти библиотеки для построения своих приложений.

На текущий момент мы прошли все механизмы для обнаружения и перехвата ошибок в эрланговых программах. В следующих главах мы используем эти механизмы для построения надёжных программных систем, которые могут восстанавливаться после сбоев. Мы закончили с программированием, рассчитанным на работу в однопроцессорных системах.

Следующая глава будет рассматривать простые распределённые системы.

1. Кроме сигнала от `exit(Pid, kill)`
2. Использование `sleep` для синхронизации опасно. В маленьких примерах это допустимо, но в рабочем коде синхронизация должна выполняться явно.
3. Когда процесс завершается нормально — это то же самое, как если бы он вычислил `exit(normal)`

Глава 10. Распределённое программирование

В этой главе мы введём понятие библиотек и примитивов Эрланга, которые мы будем использовать для написания распределённых эрланговых программ. *Распределённые программы* — это программы, которые созданы для работы в сети компьютеров и могут координировать свои действия только через передачу сообщений.

Есть ряд причин, по которым мы можем захотеть написать распределённую программу. Вот некоторые из них:

Производительность

Мы можем заставить наши программы работать быстрее, организовав работу разных частей программы параллельно на разных машинах.

Надёжность

Мы можем сделать системы устойчивыми к сбоям, спроектировав их для работы на нескольких машинах. Если одна машина выходит из строя, то мы можем продолжить работу на другой машине.

Масштабируемость

С ростом приложения мы рано или поздно исчерпаем возможности даже самой мощной машины. После этого нам придётся добавлять новые машины для наращивания вычислительной мощности. Добавление новой машины должно быть простой операцией, которая не требует больших изменений в архитектуре приложения.

Прирождённая распределённость

Многие приложения являются распределёнными по своей сути. Если мы пишем многопользовательскую игру или чат, разные пользователи будут разбросаны по всему земному шару. Если в каком-то географическом месте у нас будет много пользователей, то нам захочется разместить вычислительные ресурсы рядом с ними.

Развлечение

Большинство интересных программ, которые я пишу — распределённые. Многие из них включают взаимодействие людей и машин по всему миру.

В этой главе мы поговорим о двух основных моделях распределённости:

- *Распределённый Эрланг*: обеспечивает метод для программирования приложений, которые работают на наборе сильно связанных компьютеров. Распределённые Эрланг программы пишутся так, чтобы работать на узлах Эрланга. Мы можем порождать процессы на любом узле и все примитивы передачи сообщений и обработки ошибок, о которых мы говорили в предыдущих

главах, работают как для случая одиночного узла, так и на нескольких.

Распределённые эрланговые приложения работают в *доверенной* среде — т.к. любой узел может выполнить любую операцию на любом другом узле Эрланга, а это подразумевает высокую степень доверия. Типично распределённые эрланговые приложения работают на кусках одной локальной сети за межсетевым экраном, хотя, конечно, они могут работать и в открытой сети.

1. Например машины, находящиеся в одной локальной сети, занимающейся решением конкретной проблемы.
2. *Распределение на основе сокетов*: используя TCP/IP сокеты, можно писать распределённые приложения, которые работают в *небезопасной* среде. Программная модель менее мощная, по сравнению с распределённым Эрлангом, но более безопасная. В разделе 10.5 *Распределение на основе сокетов*, на стр. ____ мы увидим, как создавать приложения, используя простой распределённый механизм на основе сокетов.

Если вы подумаете о предыдущих главах, то вспомните, что основной строительный блок для наших программ — это процесс. Писать распределённые программы на Эрланге легко: всё, что нам надо — это порождать наши процессы на правильных машинах, а затем всё будет работать как и раньше.

Все мы привыкли к написанию последовательных программ. Написание распределённых программ обычно гораздо труднее. В этой главе мы посмотрим на ряд техник для написания простых распределённых программ. И хотя эти программы просты, они очень полезны.

А начнём мы с ряда маленьких примеров. Для них нам понадобится изучить только две вещи, а затем мы сможем создать нашу первую распределённую программу. Мы узнаем, как запускать узел Эрланга и как выполнять удалённый вызов процедуры на удалённом узле Эрланга.

Когда я разрабатываю распределённое приложение, я всегда работаю над программой в определённом порядке:

Я пишу и тестирую программу в обычной, нераспределённой сессии Эрланга. Это то, где мы были до текущего момента, так что здесь не возникнет никаких новых проблем.

Я тестирую программу на двух различных узлах Эрланга, работающих *на одном компьютере*.

Я тестирую программу на двух различных узлах Эрланга, работающих *на двух физически разделённых компьютерах*, находящихся либо в одной локальной сети,

либо где-то в Интернете.

Последний шаг может оказаться проблематичным. Если мы работаем на машинах в одном административном домене, то это редко бывает проблемой. Но когда вовлечённые узлы принадлежат машинам из разных доменов, мы можем столкнуться с проблемой связи и нам придётся обеспечить, чтобы настройки межсетевого экрана и настройки безопасности были корректными.

В следующих частях мы создадим простой сервер имён, пройдя эти шаги по-порядку. Более точно, мы сделаем следующее:

Напишем и протестируем сервер имён в обычной, нераспределённой эрланговой системе.

Протестируем сервер имён на двух узлах на одной машине.

Протестируем сервер имён на двух разных узлах на двух разных машинах в одной локальной сети.

Протестируем сервер имён на двух разных машинах, относящихся к двум разных доменам в двух разных странах.

10.1 Сервер имён

Сервер имён — это программа, которая, получив имя, возвращает значение, связанное с этим именем. Мы также можем менять значение, связанное с определённым именем.

Наш первый сервер имён чрезвычайно прост. Он не устойчив к сбоям, так что все данные, хранящиеся в нём будут потеряны при сбое. Цель этого упражнения не сделать надёжный сервер имён, а начать разбираться с техниками распределённого программирования.

Шаг 1: Простой сервер имён

Наш сервер имён kvs — это простой сервер вида ключ-значение. У него следующий интерфейс:

```
@spec kvs:start() -> true
```

Запускает сервер; создаёт сервер с зарегистрированным именем kvs.

```
@spec kvs:store(Key, Value) -> true
```

Связывает ключ и значение.

```
@spec kvs:lookup(Key) -> {ok, Value} | undefined
```

Ищет значения для ключа и возвращает `{ok, Value}`, если с ключом связано значение; в противном случае возвращает `undefined`.

Сервер ключ-значение реализуется посредством примитивов `get` и `put` для словаря процесса:

[Скачать kvs.erl](#)

```
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
    kvs ! {self(), Q},
    receive
        {kvs, Reply} ->
            Reply
    end.

loop() ->
    receive
        {From, {store, Key, Value}} ->
            put(Key, {ok, Value}),
            From ! {kvs, true},
            loop();
        {From, {lookup, Key}} ->
            From ! {kvs, get(Key)},
            loop()
    end.
```

Мы начнём с локального тестирования сервера, чтобы посмотреть — работает ли он корректно:

```
1> kvs:start().
true
2> kvs:store({location, joe}, "Stockholm").
```

```
true
3> kvs:store(weather, raining).
true
4> kvs:lookup(weather).
{ok,raining}
5> kvs:lookup({location, joe}).
{ok,"Stockholm"}
6> kvs:lookup({location, jane}).
undefined
```

Пока что у нас никаких неприятных сюрпризов.

Шаг 2: Клиент на одном узле, сервер на другом узле, но на той же машине

Теперь мы запустим два узла Эрланга на *одном* компьютере. Чтобы сделать это нам надо открыть два терминальных окна и запустить две системы Эрланга.

Первое: запускаем терминальную оболочку и в ней запускаем распределённый узел Эрланга с именем *gandalf*. Затем запускаем сервер:

```
$ erl -sname gandalf
(gandalf@localhost) 1> kvs:start().
true
```

1. *Замечание для Windows:* в системе Windows имя может оказаться не *localhost*. Если оно не *localhost*, то вам придётся использовать имя, которое Windows вернёт вместо *localhost*, во всех последующих командах.

Аргумент *-sname gandalf* означает «запустить узел Эрланга с именем *gandalf* на локальной машине». Заметьте, как оболочка Эрланга пишет имя узла Эрланга перед командной подсказкой.

1. Имя узла имеет вид *Name@Host*. *Name* и *Host* — это атомы и если они содержат какие-либо не атомные символы, то такие атомы должны быть в одинарных кавычках.

Второе: запускаем вторую терминальную сессию и запускаем узел Эрланга с именем *bilbo*. После этого мы можем вызывать функции из *kvs*, используя библиотечный модуль *rpc*. (Заметьте, что *rpc* — это стандартный модуль библиотеки Эрланга, а не то, что мы написали ранее).

```
$ erl -sname bilbo
(bilbo@localhost) 1> rpc:call(gandalf@localhost,
kvs,store, [weather, fine]).
```

```
true  
(bilbo@localhost) 2> rpc:call(gandalf@localhost,  
kvs,lookup,[weather]).  
{ok,fine}
```

Возможно это и не выглядит так, как надо, но мы только что выполнили наше первое распределённое вычисление! Сервер работал на первом узле, а клиент — на втором.

Вызов для установки переменной *weather* был сделан на узле *bilbo*. Мы можем вернуться обратно на узел *gandalf* и проверить значение *weather*:

```
(gandalf@localhost) 2> kvs:lookup(weather).  
{ok,fine}
```

`rpc:call(Node, Mod, Func, [Arg1, Arg2, ..., ArgN])` выполняет удалённый вызов процедуры на узле `Node`. Функция, которая вызывается — это `Mod:Func(Arg1, Arg2, ..., ArgN)`.

Как мы можем видеть, программа работает как для случая нераспределённого Эрланга. Единственное отличие — это то, что клиент работает на одном узле, а сервер — на другом.

Следующий шаг — это запуск клиента и сервера на разных машинах.

Шаг 3: Клиент и сервер на разных машинах в одной локальной сети

Мы будем использовать два узла. Первый узел — это `gandalf` на машине `doris.myerl.example.com` и второй узел — это `bilbo` на машине `george.myerl.example.com`. Перед тем, как сделать это, мы откроем два терминальных окна на двух разных машинах. Назовём эти окна `doris` и `george`. Когда мы это сделаем, мы сможем выполнять команды на обеих машинах.

1. Используем что-то на подобие SSH

Шаг 1: запускаем узел Эрланга на машине `doris`:

```
doris $ erl -name gandalf -setcookie abc  
(gandalf@doris.myerl.example.com) 1> kvs:start().  
true
```

Шаг 2: запускаем узел Эрланга на машине `george` и посыпаем несколько команд к `gandalf`:

```
george $ erl -name bilbo -setcookie abc
```

```
(bilbo@george.myerl.example.com) 1> rpc:call("mailto:gandalf@doris.myerl.ex  
                                kvs, store, [weather,cold]).  
true  
(bilbo@george.myerl.example.com)  
2> rpc:call("mailto:gandalf@doris.myerl.example.com", kvs, lookup, [weather  
{ok,cold}]
```

Всё ведёт себя в точности также, как для случая двух разных узлов на одной машине.

Чтобы это заработало, запуск будет чуть более сложнее, чем в случае, когда мы запускали два узла на одной машине. Сейчас нам надо сделать следующее:

1. Запустить Эрланг с параметром `-name`. Когда у нас два узла на одной машине мы используем «короткие» имена (это видно по признаку `-sname`), но если они в разных сетях, то надо использовать параметр `-name`.

Мы можем использовать `-sname` в случае, когда машины в одной подсети.

Использование `-sname` — это единственный рабочий способ при отсутствии ДНС.

1. Убедиться, что на обоих узлах одинаковые куки (`cookie`). Именно поэтому оба узла были запущены с параметром командной строки `-setcookie abc`. (Мы поговорим о куках позднее в этой главе(5))
2. Убедиться, что полное имя машин для узлов разрешается ДНС-ом. В моём случае доменное имя `myerl.example.com` полностью локальное для моей домашней сети и разрешается локально добавлением записи в файл `/etc/hosts`.
3. Убедиться, что на обеих системах одинаковые версии кода(6), который мы хотим выполнить. В нашем случае одинаковые версии кода для `kvs` должны быть доступны на обеих системах. Есть несколько способов достичь этого:
 - a) Дома у меня есть два физически раздельных компьютера без совместно используемых файлов. Здесь я физически копирую `kvs.erl` на обе машины перед запуском.
 - b) На моём рабочем компьютере у меня рабочая станция с разделяемым по NFS диском. Здесь я просто запускаю Эрланг в разделяемом каталоге с двух различных рабочих станций.
 - c) Сконфигурировать сервер кода. Здесь я не буду объяснять как это сделать. Почитайте руководство к модулю `erl_prim_loader`.
 - d) Использовать команду шелла `nl(Mod)`. Она загружает модуль `Mod` на всех подсоединеных узлах.

Замечание: чтобы это работало, надо чтобы все узлы были подсоединенены. Узлы

соединяются, когда они пытаются получить доступ друг к другу. Это происходит, когда вы впервые вычисляете выражение, включающее удалённый узел. Простейший способ сделать это — выполнить `net_adm:ping(Node)` (более подробно см. руководство по `net_adm`).

(5) Когда мы запускаем два узла на одной и той же машине, оба узла могут получить доступ к одним и тем же кукам, `$HOME/.erlang.cookie`, поэтому мы не добавляли куки к командной строки Erlang.

(6) Желательно, чтобы одинаковым была `b` версия Erlang. Если вы не сделаете этого, вы можете получить серьезные и непредсказуемые ошибки.

Шаг 4: Клиент и сервер на разных машинах в Интернете

В принципе, это то же самое, что и шаг 3, но сейчас нам надо гораздо больше позаботиться о безопасности. Когда мы запускаем два узла в одной локальной сети, нам, возможно, не надо сильно волноваться по поводу безопасности. В большинстве организаций локальная сеть отделена от Интернета межсетевым экраном. За этим экраном мы вольны выбирать IP адреса совершенно наобум и конфигурировать наши машины сколь угодно криво.

Когда же мы подключаем несколько машин эрлангового кластера к Интернету, мы можем ожидать проблем от межсетевых экранов, которые не пропускают входящие соединения. Нам нужно правильно сконфигурировать наши межсетевые экраны для приёма входящих соединений. Общей рекомендации как это сделать, не существует, т. к. все сетевые экраны различны.

Чтобы подготовить вашу систему к распределённому Эрлангу вам нужно выполнить следующие шаги:

Убедитесь, что порт 4369 открыт для TCP и UDP трафика. Этот порт используется программой `epmd` (сокращение от *Erlang Port Mapper Daemon*).

Выбрать порт или диапазон портов для использования в распределённом Эрланге и убедиться, что эти порты открыты. Если эти порты от `Min` до `Max` (используйте `Min=Max`, если хотите использовать только один порт), то запускайте Эрланг следующей командой:

```
$ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \
          inet_dist_listen_max Max
```

10.2 Примитивы распределения

Центральная концепция в распределённом Эрланге — это узел. Узел — это самодостаточная система Эрланга, содержащая полную виртуальную машину со своим собственным адресным пространством и собственным набором процессов.

Доступ к одиночному узлу или набору узлов обезопасен посредством кук (*cookie*). У каждого узла свои куки и эти куки должны быть одинаковы для всех узлов, с которыми наш узел собирается общаться. Чтобы обеспечить это, все узлы в распределённой эрланговой системе должны быть запущены с одинаковыми куками или должны установить свои куки в одинаковое значение вызовом `erlang:set_cookie`.

Набор соединённых узлов, имеющих одинаковые куки, образует эрланговый кластер.

Следующие встроенные функции (BIF) используются для написания распределённых программ.

1. Для более детального описания этих BIF-ов, смотрите руководство для модуля Erlang.

```
@spec spawn(Node, Fun) -> Pid
```

Работает в точности, как `spawn(Fun)`, только новый процесс порождается на узле `Node`.

```
@spec spawn(Node, Mod, Func, ArgList) -> Pid
```

Работает в точности, как `spawn(Mod, Func, ArgList)`, только новый процесс порождается на узле `Node`. `spawn(Mod, Func, Args)` создаёт новый процесс, который вычисляет `apply(Mod, Func, Args)`. Он возвращает PID нового процесса.

Замечание: эта форма порождения более надёжна, чем `spawn(Node, Fun)`. `spawn(Node, Fun)` может сломаться, если на распределённых узлах работают хотя бы малость отличающиеся версии соответствующего модуля.

```
@spec spawn_link(Node, Fun) -> Pid
```

Работает в точности, как `spawn_link(Fun)`, только новый процесс порождается на узле `Node`.

```
@spec spawn_link(Node, Mod, Func, ArgList) -> Pid
```

Работает в точности, как `spawn(Node, Mod, Func, ArgList)`, только новый процесс связывается с текущим процессом.

```
@spec disconnect_node(Node) -> bool() | ignored
```

Принудительно отсоединяет узел.

```
@spec monitor_node(Node, Flag) -> true
```

Если `Flag` имеет значение `true`, то включается мониторинг. Если `Flag` имеет значение `false`, то мониторинг выключается. При включенном мониторинге процессу, который выполнил эту функцию, посылаются сообщения `{nodeup, Node}` и `{nodedown, Node}` в случае, когда узел `Node` присоединяется или покидает набор подключенных узлов Эрланга.

```
@spec node() -> Node
```

Возвращает имя локального узла. Если узел не является распределённым, то возвращается `nonode@nohost`.

```
@spec node(Arg) -> Node
```

Возвращает узел, где находится `Arg`. `Arg` может быть `PID`, ссылка или порт. Если узел не является распределённым, то возвращается `nonode@nohost`.

```
@spec nodes() -> [Node]
```

Возвращает список всех других узлов в сети, с которыми мы соединены.

```
@spec is_alive() -> bool()
```

Возвращает `true`, если локальный узел жив и может быть частью распределённой системы. В противном случае возвращает `false`.

Дополнительно, для отправки сообщений к процессу, зарегистрированному локально в наборе распределённых узлов Эрланга может использоваться `send`. Синтакс этого следующий:

```
{RegName, Node} ! Msg
```

посыпает сообщение `Msg` к зарегистрированному на узле `Node` процессу `RegName`.

Пример удалённого порождения

Как простой пример, мы можем показать — как порождать процесс на удалённом узле. Начнём со следующей программы:

[Скачать dist_demo.erl](#)

```
-module(dist_demo).
-export([rpc/4, start/1]).

start(Node) ->
    spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
    Pid ! {rpc, self(), M, F, A},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {rpc, Pid, M, F, A} ->
            Pid ! {self(), (catch apply(M, F, A))},
            loop()
    end.
```

Затем мы запускаем два узла, причём оба узла должны быть способны загрузить этот код. Если оба узла запущены на одной машине, тогда это не проблема. Мы просто запускаем два Эрланга из одного и того же каталога. Если же узлы находятся на физически распределённых машинах с разными файловыми системами, то программу надо скопировать на все машины и скомпилировать перед запуском обоих узлов (ну или можно скопировать `.beam` файл на все машины). В этом примере я полагаю, что это уже сделано.

На машине `doris` мы запускаем узел под названием `gandalf`:

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1>
```

А на машине `george` мы запускаем узел под названием `bilbo`, помня об использовании тех же кук:

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1>
```

Теперь (на `bilbo`) мы можем породить процесс на удалённом узле (`gandalf`):

```
(bilbo@george.myerl.example.com) 1> Pid =  
    dist_demo:start('gandalf@doris.myerl.example.com').  
<0094.40.0>
```

`Pid` — это идентификатор процесса на удалённом узле и теперь мы можем вызвать `dist_demo:rpc/4` для выполнения удалённого вызова процедуры на удалённом узле:

```
(bilbo@george.myerl.example.com)2> dist_demo:rpc(Pid, erlang, node, []).  
'gandalf@doris.myerl.example.com'
```

Это выполняет `erlang:node()` на удалённом узле и возвращает значение.

10.3 Библиотеки для распределённого программирования

Предыдущая часть показала встроенные функции, которые мы можем использовать для написания распределённых программ. Фактически, большинство программистов на Эрланге никогда не используют эти функции. Вместо этого они используют ряд мощных библиотек для распределения. Библиотеки написаны с использованием этих функций, но они скрывают большинство сложностей от программиста.

Два модуля из стандартной поставки покрывают большинство нужд:

- `rpc` обеспечивает ряд сервисов удалённого вызова процедур;
- в `global` есть функции для регистрации имён и блокировок в распределённой системе и для поддержки полностью соединённой сети.

Читайте руководство по RPC

Модуль `rpc` — это настоящий рог изобилия функциональности

Одна наиболее полезная функция из модуля `rpc` — следующая:

```
call(Node, Mod, Function, Args) -> Result | {badrpc, Reason}
```

Она выполняет `apply(Mod, Function, Args)` на узле `Node` и возвращает результат `Result` или `{badrpc, Reason}`, в случае неуспеха.

10.4 Защита с помощью кук

Для того, чтобы два распределённых узла Эрланга могли общаться между собой, им надо иметь одинаковые куки (*magic cookie*). Мы можем установить куки тремя способами:

- **Способ 1:** сохранить одинаковые куки в файле `$HOME/.erlang.cookie`. Этот файл содержит строку случайных данных и создаётся автоматически, когда Эрланг запускается на вашей машине в первый раз.

Этот файл можно скопировать на все машины, которые будут участвовать в сеансе распределённого Эрланга. Или же мы можем явно установить значение. К примеру, на Linux мы можем выполнить следующие команды:

```
$ cd  
$ cat > .erlang.cookie  
AFRTY12ESS3412735ASDF12378  
$ chmod 400 .erlang.cookie
```

Команда `chmod` делает файл `.erlang.cookie` доступным только владельцу файла.

- **Способ 2:** когда запускается Эрланг мы можем использовать параметр командной строки `-setcookie C`, чтобы установить значение куков в `C`. Например:

```
$ erl -setcookie AFRTY12ESS3412735ASDF12378 ...
```

- **Способ 3:** встроенная функция `erlang:set_cookie(node(), C)` устанавливает куки на локальном узле в атом `C`.

Замечание: если ваше окружение небезопасно, то способы 1 и 3 предпочтительнее по сравнению со способом 2, т.к. на UNIX любой может узнать вашу куку, используя команду `ps`.

Если вам любопытно, то куки никогда не передаются по сети в открытом виде. Куки используются только для первоначальной аутентификации сеанса. Сеансы распределённого Эрланга не шифруются, но они могут быть запущены поверх шифрованных каналов. (Поиските на Гугле более современную информацию из списка рассылки Эрланга).

10.5 Распределение на основе сокетов

В этой части мы напишем простую программу, используя распределение, основанное

на сокетах. Как мы уже видели, распределённый Эрланг хорош для написания кластерных приложений, где вы можете доверять всем присутствующим, но не очень подходит для открытой среды, где нельзя доверять первому встречному.

Основная проблема с распределённым Эрлангом — это то, что клиент может породить любой процесс на серверной машине. Так что, для полного разрушения файловой системы всё, что вам надо сделать — это выполнить следующее:

```
rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *"])
```

Распределённый Эрланг хорош в случае, когда все машины ваши и вы хотите управлять ими из одного места. Однако эта модель вычислений не подходит для случая, когда разные машины принадлежат разным людям и они хотят иметь контроль над тем, какие программы будут выполняться на их машинах.

В таких обстоятельствах мы будем использовать ограниченную версию порождения, когда у владельца определённой машины есть контроль над тем, что запускается на его машине.

lib_chan

`lib_chan` — это модуль, который позволяет пользователю явно управлять тем, какие процессы порождаются на его машине. Реализация `lib_chan` достаточно сложна, так что я не буду излагать её здесь. Вы можете найти её в Приложении D на стр. _____. Интерфейс у неё следующий:

```
@spec start_server() -> true
```

Запускает сервер на локальной машине. Поведение сервера определяется содержимым файла `$HOME/.erlang/lib_chan.conf`.

```
@spec start_server(Conf) -> true
```

Запускает сервер на локальной машине. Поведение сервера определяется содержимым файла Conf.

В обоих случаях в файле конфигурации сервера находятся кортежи следующего вида:

```
{port, NNNN}
```

Запускает приём соединений на порту NNNN

```
{service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS}
```

Сервис S защищается паролем P . При запуске сервиса для обработки сообщений от клиента создаётся процесс посредством порождения $SomeMod:SomeFunc(MM, ArgsC, SomeArgsS)$. Здесь MM — это PID процесса-посредника, который используется для передачи сообщений клиенту, а параметр $ArgC$ приходит из клиентского вызова на подключение (к серверу).

```
@spec connect(Host, Port, S, P, ArgsC) -> {ok, Pid} | {error, Why}
```

Пытается открыть порт $Port$ на машине $Host$ и затем пытается активировать сервис S , который защищён паролем P . Если пароль верный, то возвращается $\{ok, Pid\}$, где Pid — это идентификатор процесса-посредника используемого для передачи сообщений к серверу.

Когда соединение устанавливается из клиента посредством вызова $connect/5$, создаются два процесса-посредника: один на стороне клиента и один на стороне сервера. Эти процессы организуют преобразование сообщений Эрланга в пакеты данных TCP, перехватывая при этом выходы управляющих процессов и закрытие сокета.

Это объяснение может показаться сложным, но оно окажется гораздо более простым, когда мы начнём его использовать.

Далее идёт полный пример использования *lib_chan* совместно с описанным ранее сервисом *kvs*.

Код сервера

Для начала напишем файл конфигурации:

```
{port, 1234} .  
{service, nameServer, password, "ABXy45" ,  
 mfa, mod_name_server, start_me_up, notUsed} .
```

Это означает, что мы собираемся предлагать сервис, называемый *nameServer* на порту 1234 нашей машины. Сервис защищается паролем ABXy45.

Когда с клиента устанавливается соединение посредством вызова

```
connect(Host, 1234, nameServer, "ABXy45", nil)
```

сервер порождает `mod_name_server:start_me_up(MM, nil, notUsed)` (так в pdf-оригинале), где `MM` — это `PID` процесса-посредника, который используется для общения с клиентом.

Важно: сейчас вы должны глазеть на предыдущую строку кода и пытаться понять — откуда же взялись аргументы для этого вызова:

- `mod_name_server`, `start_me_up` и `notUsed` взяты из файла конфигурации
- `nil` — это последний аргумент в вызове `connect`.

Модуль `mod_name_server` выглядит так:

[Скачать mod_name_server.erl](#)

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) ->
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! {send, kvs:lookup(K)},
            loop(MM);
        {chan_closed, MM} ->
            true
    end.
```

`mod_name_server` работает по следующему протоколу:

- если клиент посыпает серверу сообщение `{send, X}`, то оно появится в `mod_name_server` как сообщение вида `{chan, MM, X}` (`MM` — это `PID` серверного процесса-посредника).
- если клиент завершается или сокет, используемый для связи, закрывается по какой-либо причине, то сервер получает сообщение вида `{chan_closed, MM}`.
- если сервер хочет послать сообщение `X` клиенту, он делает это посредством вызова `MM ! {send, X}`.
- если сервер хочет закрыть соединение явно, он делает это, выполняя `MM ! close`.

Этот протокол — протокол посредника, которому подчиняются как клиентский, так и серверный код. Код сокета для посредника объясняется более подробно в части D.2,

`lib_chan_mm`: Постройте, на стр. ____.

Чтобы протестировать этот код, мы сначала убедимся, что он работает правильно на одной машине.

Запускаем сервер имён (и модуль kvs):

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 1234...
true
```

После этого мы можем запустить второй сеанс Эрланга и протестировать всё это со стороны клиента:

```
1> {ok, Pid} = lib_chan:connect("localhost", 1234, nameServer,
"ABXy45", "").
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```

Проверив, что это работает на одной машине, мы проходим те же описанные ранее шаги и выполняем подобные тесты на двух физически разделённых машинах.

Заметьте, что в этом случае содержимое конфигурационного файла определяется владельцем удалённой машины. Файл конфигурации указывает какие приложения разрешены на этой машине и какой порт должен использоваться для связи с этими приложениями.

Глава 11. Лёгкий IRC

Вот и пришло время для приложения. До сих пор мы видели только разрозненные части. Мы видели как писать последовательный код, как порождать процессы, как регистрировать процессы и т.д. Теперь мы соберём все эти части в одно работающее целое.

В этой главе мы создадим простую IRC-подобную программу. Мы не будем придерживаться настоящего IRC протокола. Вместо этого мы придумаем наш

собственный совершенно другой и не совместимый ни с чем протокол. С точки зрения пользователя наша программа является реализацией IRC, хотя нижележащая реализация гораздо проще, чем это могло бы ожидаться, т. к. мы используем сообщения Эрланга как основу для межпроцессорных сообщений. Это полностью устраниет разбор сообщений и значительно упрощает дизайн.

Наша программа является программой на чистом Эрланге, которая совершенно не использует библиотеки OTP и минимально использует стандартные библиотеки. Так что, к примеру, у неё полностью самодостаточная клиент-серверная архитектура и форма восстановления после ошибок, основанная на явном манипулировании связей. Причина неиспользования библиотек в том, что я хочу вносить вам на рассмотрение по одной концепции за раз и показывать, что мы можем достичь с одним языком и минимальным использованием библиотек. Мы будем писать код как набор компонентов. Каждый компонент прост, но вместе они работают достаточно сложно. Мы можем заставить большую часть сложностей убраться, используя библиотеки OTP, так что позднее в этой книге мы покажем более правильные способы организации кода, основанном на общих библиотеках OTP для построения деревьев клиент-серверов и наблюдения (супервизоров).

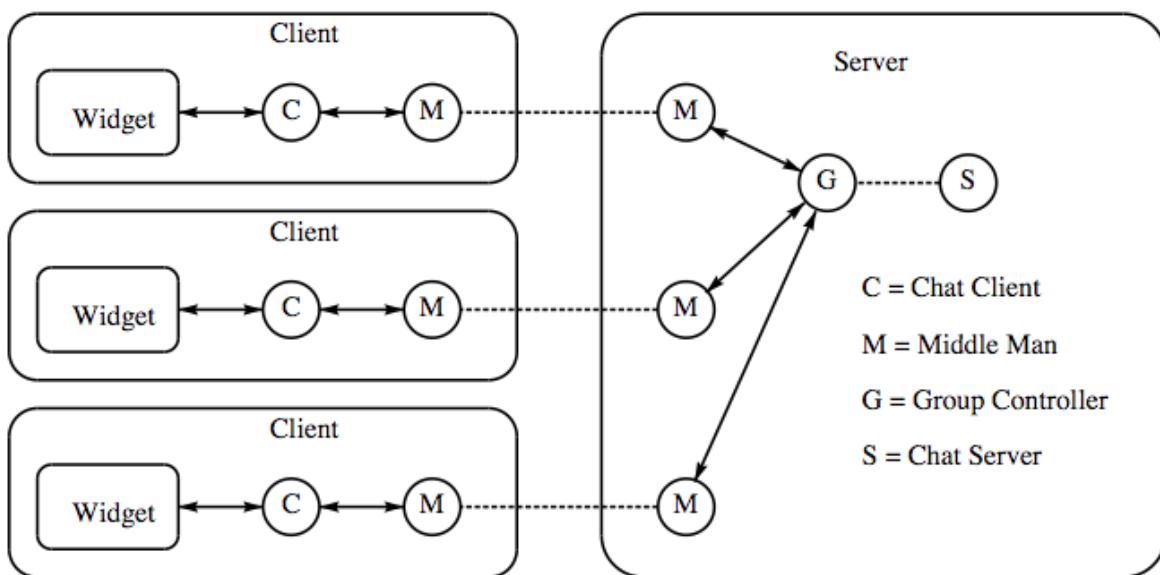


Рис.11. 1: Структура процесса

Наше приложение построено из пяти компонентов. Структура этих компонентов показана на Рис.11. 1. Рисунок показывает три клиентских узла (предполагается, что они на других машинах) и один серверный узел (тоже на другой машине). Эти компоненты выполняют следующие функции:

Интерфейс с пользователем — это графическое приложение, которое используется для отправки сообщений и отображения полученных сообщений. Сообщения

отправляются чат-клиенту.

Чат-клиент («С» на рисунке) — разбирается с сообщениями от пользовательского приложения и отправляет их к контроллеру группы для текущей группы. Принимает сообщения от контроллера группы и отправляет их к пользовательскому приложению.

Контроллер группы («G» на рисунке) — управляет одной чат-группой. Если контроллеру посыпается сообщение, то он рассыпает это сообщение всем участникам в данной группе. Он отслеживает новых участников, которые присоединились к группе и участников, которые покинули группу. Контроллер завершается, если в группе не осталось участников.

Чат-сервер («S» на рисунке) — отслеживает контроллеров группы. Чат-сервер нужен только когда новый участник пытается присоединиться к группе. Чат-сервер существует в единственном экземпляре, в то время как контроллеры групп создаются для каждой активной группы.

Посредник («М» на рисунке) — обеспечивает транспортировку данных в системе. Если процесс С посыпает сообщение к М, оно попадёт к G (см. Рис.11. 1). Процесс М скрывает низкоуровневый интерфейс сокетов между двумя машинами. Главным образом процесс М прячет физическую границу между машинами за какой-то абстракцией. Это значит, что на основе передачи сообщений Эрланга можно построить целое приложение и не заботиться о подробностях нижележащей инфраструктуры связи.

11.1 Диаграммы последовательности сообщений

Если у нас много параллельных процессов, то очень легко потерять нить происходящего. Чтобы помочь нам понять, что происходит, мы можем нарисовать диаграмму последовательности сообщений (MSD), которая показывает взаимодействие между различными процессами.

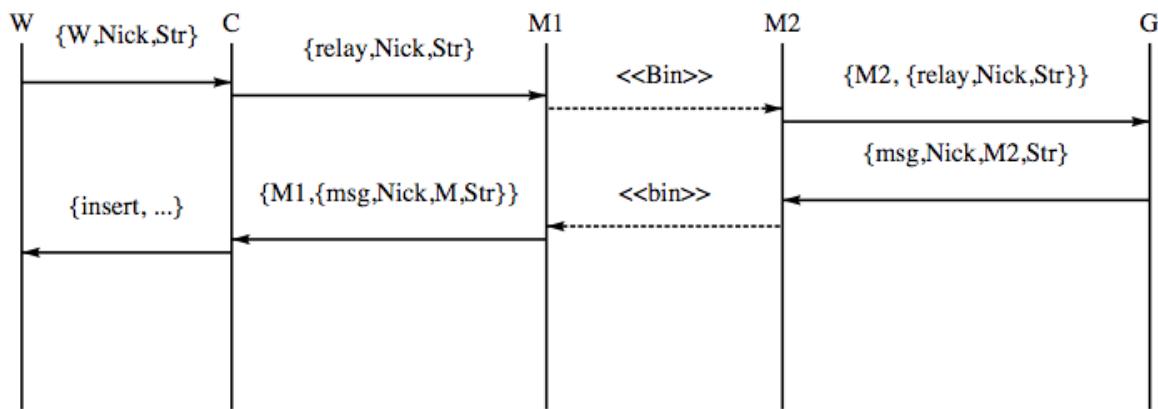


Рис.11. 2: Прохождение сообщений, участвующих в передаче текстового сообщения

Диаграмма последовательности сообщений на Рис.11. 2 показывает последовательность сообщений, которые пересылаются, когда пользователь напечатает строку в поле ввода. Это приводит к отправке сообщения к чат-контроллеру (C), за которым следует сообщение к одному из посредников (M1), затем через M2 к контроллеру группы (G). На этапе между посредниками происходит двоичное кодирование сообщений Эрланга.

MSD даёт хорошее представление того, что происходит. Если вы будете глязеть на MSD и на код программы достаточно долго, то вы сможете убедить себя в том, что этот код реализует именно ту последовательность передачи сообщений, которая изображена на диаграмме.

Когда я проектирую программу наподобие чата, я часто рисую множество MSD диаграмм — это помогает мне думать о том, что происходит. Я не большой любитель графических методов проектирования, но MSD диаграммы полезны для отображения того, что происходит в ряде параллельных процессов, которые обмениваются сообщениями для решения определённой проблемы.

А сейчас посмотрим на индивидуальные компоненты.

11.2 Пользовательский интерфейс

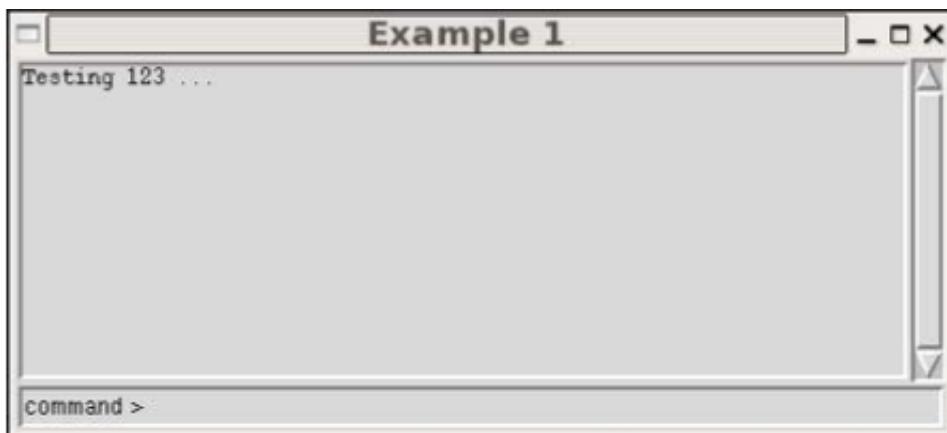


Рис.11. 3: Виджет ввода-вывода

Пользовательский интерфейс построен на базе простого виджета ввода-вывода. Этот виджет показан на Рис.11. 3. Код этого виджета достаточно длинный и в основном касается доступа к оконной системе посредством стандартной библиотеки gs. Т. к. мы пока не хотим прыгать в эту кроличью нору, то мы не покажем здесь соответствующий код (хотя вы найдёте этот код, начиная со страницы 17). Интерфейс у виджета ввода-вывода следующий:

```
@spec io_widget:start(Pid) -> Widget
```

Создаёт новый виджет ввода-вывода. Возвращает `Widget`, который является `PID`, который может использоваться для общения с виджетом. Когда пользователь печатает что-либо в поле ввода виджета процессу, который вызвал эту функцию посылаются сообщения вида `{Widget, State, Parse}`. `State` — это переменная состояния, сохранённая в виджете, которая может устанавливаться пользователем. `Parse` — это результат разбора строки ввода пользовательским парсером.

```
@spec io_widget:set_title(Widget, Str)
```

Устанавливает заголовок в виджете.

```
@spec io_widget:set_state(Widget, State)
```

Устанавливает состояние виджета.

```
@spec io_widget:insert_str(Widget, Str)
```

Вставляет строку в основную область виджета.

```
@spec io_widget:set_handler(Widget, Fun)
```

Устанавливает парсер виджета в `Fun` (см. далее).

Виджет ввода-вывода может генерировать следующие сообщения:

```
{Widget, State, Parse}
```

Это сообщение отправляется, когда пользователь вводит строку в нижней области команд виджета. `Parse` — это результат разбора этой строки парсером, связанным с данным виджетом.

```
{Widget, destroyed}
```

Это сообщение отправляется, когда пользователь разрушает виджет посредством закрытия окна.

В общем, виджет ввода-вывода — это программируемая штучка. С ним можно связать парсер, который будет использоваться для разбора всех сообщений, которые вводятся в поле ввода виджета. Разбор делается вызовом функции `Parse(Str)`. Эта функция может быть установлена вызовом `set_handler(Widget, Parse)`.

Парсер по-умолчанию — это такая функция:

```
Parse(Str) -> Str end.
```

11.3 Клиентская часть

Клиентская часть программы чата состоит из трёх процессов: виджет ввода-вывода (о котором мы уже говорили), клиент чата (который организует взаимодействие между виджетом и посредником) и процесс посредника. В этой части мы сосредоточимся на клиенте чата.

Клиент чата

Мы запускаем чат-клиент вызовом `start/0`:

Скачать [socket_dist/chat_client.erl](#)

```
start() ->
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "joe" ).
```

Он пытается подсоединиться к `localhost` на порт `2223` (это жестко закрыто в код для тестовых целей). Функция `connect/5` просто создаёт параллельный процесс, вызывая `handler/5`. А вот обработчику приходится выполнять несколько задач:

- он делает себя системным процессом, так что теперь он может перехватывать сигналы выхода
- он создаёт виджет ввода-вывода и устанавливает подсказку и заголовок этого виджета
- он порождает процесс соединения (который пытается соединиться с сервером)
- в конце он ждёт события соединения в `disconnected/2` (прим. перев.: «Синее, а не бурое! А по описанию -- бурое, а не синее!..» (C) АБС)

Код для него:

[Скачать socket_dist/chat_client.erl](#)

```
connect(Host, Port, HostPsw, Group, Nick) ->
    spawn(fun() -> handler(Host, Port, HostPsw, Group, Nick) end).

handler(Host, Port, HostPsw, Group, Nick) ->
    process_flag(trap_exit, true),
    Widget = io_widget:start(self()),
    set_title(Widget, Nick),
    set_state(Widget, Nick),
    set_prompt(Widget, [Nick, " > "]),
    set_handler(Widget, fun parse_command/1),
    start_connector(Host, Port, HostPsw),
    disconnected(Widget, Group, Nick).
```

В отключенном состоянии процесс либо получит сообщение `{connected, MM}` (2), после чего он посыпает сообщение `login` к серверу и ждёт ответа на логин, либо виджет может быть разрушен, что приводит к всеобщему завершению.

Соединяющийся процесс периодически шлёт сообщения о состоянии к чат-клиенту. Эти сообщения сразу же пересыпаются к виджету ввода-вывода для показа.

[Скачать socket_dist/chat_client.erl](#)

```
disconnected(Widget, Group, Nick) ->
    receive
        {connected, MM} ->
            insert_str(Widget, "connected to server\\nsending data\\n" ),
            MM ! {login, Group, Nick},
            wait_login_response(Widget, MM);
        {Widget, destroyed} ->
```

```

        exit(died);
{status, S} ->
    insert_str(Widget, to_str(S)),
    disconnected(Widget, Group, Nick);
Other ->
    io:format("chat_client disconnected unexpected:\~p\~n" ,[Other])
    disconnected(Widget, Group, Nick)
end.

```

Сообщение `{connected, MM}` очевидно должно прийти от соединяющегося процесса, который был создан вызовом `start_connection(Host, Port, HostPsw)`. Этот вызов создаёт параллельный процесс, который в свою очередь периодически пытается соединиться с IRC сервером.

[Скачать socket_dist/chat_client.erl](#)

```

start_connector(Host, Port, Pwd) ->
    S = self(),
    spawn_link(fun() -> try_to_connect(S, Host, Port, Pwd) end).

try_to_connect(Parent, Host, Port, Pwd) ->
    %% Parent is the Pid of the process that spawned this process
    case lib_chan:connect(Host, Port, chat, Pwd, []) of
        {error, _Why} ->
            Parent ! {status, {cannot, connect, Host, Port}},
            sleep(2000),
            try_to_connect(Parent, Host, Port, Pwd);
        {ok, MM} ->
            lib_chan_mm:controller(MM, Parent),
            Parent ! {connected, MM},
            exit(connectorFinished)
    end.

```

`try_to_connect` зацикливается навечно, пытаясь каждые две секунды подключиться к серверу. Если подключиться не удаётся, то он посылает сообщение о состоянии к чат-клиенту.

Замечание: в `start_connection` мы написали следующее:

```

S = self(),
spawn_link(fun() -> try_to_connect(S, ...) end)

```

Это не то же самое, что и здесь:

```
spawn_link(fun() -> try_to_connect(self(), ...) end)
```

Причина в том, что в первом фрагменте кода `self()` выполняется внутри родительского процесса. Во втором куске кода `self()` выполняется внутри порождённой функции, так что он возвращает идентификатор порождённого процесса, а не `PID` текущего процесса, как вы могли бы подумать. Это довольно распространённая причина для ошибок (и непонимания).

Если соединение установлено, то он посыпает сообщение `{connected, MM}` к чат-клиенту. По прибытии этого сообщения чат-клиент посыпает сообщение для логина к серверу (оба этих события происходят в `disconnected/2`) и ждёт ответа в `wait_login_response/2`:

[Скачать socket_dist/chat_client.erl](#)

```
wait_login_response(Widget, MM) ->
    receive
        {MM, ack} ->
            active(Widget, MM);
        Other ->
            io:format("chat_client login unexpected:\`~p\`~n" ,[Other]),
            wait_login_response(Widget, MM)
    end.
```

Если всё идёт по плану, то процесс должен получить подтверждающее сообщение `ack`. (В нашем случае это единственно возможный ответ, т. к. пароль точно был правильным). После получения подтверждения эта функция вызывает `active/2`:

[Скачать socket_dist/chat_client.erl](#)

```
active(Widget, MM) ->
    receive
        {Widget, Nick, Str} ->
            MM ! {relay, Nick, Str},
            active(Widget, MM);
        {MM,{msg,From,Pid,Str}} ->
            insert_str(Widget, [From,"@" ,pid_to_list(Pid)," " , Str, "\n"],
            active(Widget, MM));
        {'EXIT',Widget,windowDestroyed} ->
            MM ! close;
        {close, MM} ->
            exit(serverDied);
    Other ->
        io:format("chat_client active unexpected:\`~p\`~n" ,[Other]),
```

```
    active(Widget, MM)
end.
```

`active/2` просто шлёт сообщения от виджета к группе (и наоборот) и отслеживает соединение с группой.

За исключением некоторых объявлений модулей и простейших процедур форматирования и разбора это завершает чат-клиент.

Полный код чат-клиента приведён на стр. [_____](#)

11.4 Серверная часть

Серверная часть программы сложнее, чем клиентская. Для каждого клиента чата есть соответствующий чат-контроллер, который организует взаимодействие чат-клиента с чат-сервером. Есть единственный чат-сервер, который знает обо всех сессиях чата в данный момент и ещё есть некоторое количество менеджеров групп (по одному на чат-группу), которые управляют отдельными чат-группами.

Чат-контроллер

Чат-контроллер — это дополнение (plug-in) для `lib_chan`, дистрибутивному набору, основанному на сокетах. Мы встречали его в главе 10.5, `lib_chan`, на стр. [_____](#). `lib_chan` нуждается в конфигурационном файле и модуле дополнении.

Конфигурационный файл для системы чата следующий:

[Скачать socket_dist/chat.conf](#)

```
{port, 2223}.
{service, chat, password, "AsDT67aQ", mfa, mod_chat_controller, start, []}.
```

Если вы посмотрите назад на код `chat_client.erl`, вы увидите, что номер порта, имя сервиса и пароль согласуются с информацией из конфигурационного файла.

Модуль чат-контроллера очень прост:

[Скачать socket_dist/mod_chat_controller.erl](#)

```
-module(mod_chat_controller).
-export([start/3]).
-import(lib_chan_mm, [send/2]).
```

```

start(MM, _, _) ->
    process_flag(trap_exit, true),
    io:format("mod_chat_controller off we go ...~p~n" ,[MM]),
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, Msg} ->
            chat_server ! {mm, MM, Msg},
            loop(MM);
        {'EXIT', MM, _Why} ->
            chat_server ! {mm_closed, MM};
        Other ->
            io:format("mod_chat_controller unexpected message =~p (MM=~p)
[Other, MM]),",
            loop(MM)
    end.

```

Этот код будет принимать только два сообщения. Когда клиент соединяется он получит произвольное сообщение и просто отправит его к чат-серверу. С другой стороны, если сеанс завершается по какой-либо причине, он получит сообщение о выходе и затем скажет чат-серверу, что клиент умер.

Чат-сервер

Чат-сервер — это зарегистрированный процесс, называемый (что неудивительно) `chat_server`. Вызов `chat_server:start/0` запускает и регистрирует сервер, а он запускает `lib_chan`.

[Скачать socket_dist/chat_server.erl](#)

```

start() ->
    start_server(),
    lib_chan:start_server("chat.conf" ).

start_server() ->
    register(chat_server,
    spawn(fun() ->
        process_flag(trap_exit, true),
        Val = (catch server_loop([])),
        io:format("Server terminated with:~p~n" ,[Val])
    end)).

```

Серверный цикл прост. Он ждёт сообщения `{login, Group, Nick}`(3) от посредника

с `PID`, равным `Channel`. Если есть контроллер чат-группы для этой группы, то он просто посыпает сообщение о логине к контроллеру группы, а иначе он запускает нового контроллера группы.

Чат-сервер — это единственный процесс, который знает `PID`-ы всех контроллеров групп, так что, когда делается новое соединение к системе, к чат-серверу приходит запрос на поиск идентификатора процесса контроллера группы.

Сам по себе сервер прост:

Скачать [socket_dist/chat_server.erl](#)

```
server_loop(L) ->
    receive
        {mm, Channel, {login, Group, Nick}} ->
            case lookup(Group, L) of
                {ok, Pid} ->
                    Pid ! {login, Channel, Nick},
                    server_loop(L);
                error ->
                    Pid = spawn_link(fun() ->
                        chat_group:start(Channel, Nick)
                    end),
                    server_loop([{Group,Pid}|L])
            end;
        {mm_closed, _} ->
            server_loop(L);
        {'EXIT', Pid, allGone} ->
            L1 = remove_group(Pid, L),
            server_loop(L1);
        Msg ->
            io:format("Server received Msg=~p~n", [Msg]),
            server_loop(L)
    end.
```

Код для манипуляций списком групп включает в себя несколько простых подпрограмм для обработки списков:

Скачать [socket_dist/chat_server.erl](#)

```
lookup(G, [{G,Pid}|_]) -> {ok, Pid};
lookup(G, [_|T])          -> lookup(G, T);
lookup(_, [])              -> error.

remove_group(Pid, [{G,Pid}|T]) -> io:format("~p removed~n", [G]), T;
```

```
remove_group(Pid, [H|T])      -> [H|remove_group(Pid, T)];  
remove_group(_, [])           -> [].
```

Менеджер группы

К текущему моменту всё, что осталось — это менеджер группы. Важнейшая часть этого — диспетчер.

[Скачать socket_dist/chat_group.erl](#)

```
group_controller([]) ->  
    exit(allGone);  
group_controller(L) ->  
    receive  
        {C, {relay, Nick, Str}} ->  
            foreach(fun({Pid, _}) -> Pid ! {msg, Nick, C, Str} end, L),  
            group_controller(L);  
        {login, C, Nick} ->  
            controller(C, self()),  
            C ! ack,  
            self() ! {C, {relay, Nick, "I'm joining the group" }},  
            group_controller([{C, Nick}|L]);  
        {close,C} ->  
            {Nick, L1} = delete(C, L, []),  
            self() ! {C, {relay, Nick, "I'm leaving the group" }},  
            group_controller(L1);  
        Any ->  
            io:format("group controller received Msg=~p~n" , [Any]),  
            group_controller(L)  
    end.
```

Аргумент `L` в `group_controller(L)` — это список имён и идентификаторов процессов посредников `{Pid, Nick}`.

Когда менеджер группы получает сообщение `{relay, Nick, Str}`, он просто рассыпает его всем процессам в группе. Если приходит сообщение `{login, C, Nick}`, он добавляет кортеж `{C, Nick}` в список рассылки. Важно упомянуть вызов `lib_chan_mm:controller/2`. Этот вызов устанавливает управляющий процесс посредника в контроллер группы, что означает, что *все сообщения к сокету, управляемому посредником, будут посланы к контроллеру группы* — это, вероятно, главная часть для понимания — как работает весь этот код.

Всё, что остаётся — это код, который запускает сервер группы:

[Скачать socket_dist/chat_group.erl](#)

```

-module(chat_group).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [foreach/2, reverse/2]).
-export([start/2]).

start(C, Nick) ->
    process_flag(trap_exit, true),
    controller(C, self()),
    C ! ack,
    self() ! {C, {relay, Nick, "I'm starting the group" }},
    group_controller([{C,Nick}]).
```

и функция `delete/3`, вызываемая из цикла диспетчера процесса

[Скачать socket_dist/chat_group.erl](#)

```

delete(Pid, [{Pid,Nick}|T], L) -> {Nick, reverse(T, L)};
delete(Pid, [H|T], L)           -> delete(Pid, T, [H|L]);
delete(_, [], L)               -> {"????", L}.
```

11.5 Запуск приложения

Приложение целиком располагается в каталоге `path/to/code/socket_dist`. Оно также использует некоторые библиотечные модули из каталога `path/to/code`.

Для запуска приложения получите исходные коды веб-сайта этой книги и распакуйте их в какой-нибудь каталог. (Здесь мы предполагаем, что это каталог `/home/joe/erlbook`). Откройте окно терминала и выполните следующие команды:

```

$ cd /home/joe/erlbook/code
/home/joe/erlbook/code $ make
...
/home/joe/erlbook/code $ cd socket_dist
/home/joe/erlbook/code/socket_dist $ make chat_server
... 
```

Это запустит чат-сервер. А теперь нам надо открыть другое терминальное окно и запустить тест клиента:

```

$ cd /home/joe/erlbook/code/socket_dist
/home/joe/erlbook/code/socket_dist $ make chat_client
... 
```

Запуск `make chat_client` выполняет функцию `chat_client:test()`. Это на самом деле создаёт четыре окна, которые подключаются к тестовой группе, названной «general». На Рис.11. 4 мы можем увидеть снимок экрана, показывающий как выглядит система после выдачи этих команд.



Рис.11. 4: Снимок экрана, показывающий четыре окна, подключенные к одной группе

Для развертывания системы в Интернете всё, что нам надо сделать — это поменять пароль и порт на что-нибудь подходящее и разрешить входящие соединения на порт, который мы выбрали.

11.6 Исходные коды программы чата

Итак, мы завершили описание программы чата. При описании программы мы разбили её на несколько маленьких фрагментов и опустили некоторую часть кода. Этот раздел содержит весь код в одном месте, что облегчает его чтение. Если у вас есть трудности с пониманием, что делает та или иная часть кода, обратитесь к описанию, изложенному ранее в этой главе.

Чат-клиент

[Скачать socket_dist/chat_client.erl](#)

```
-module(chat_client).
```

```

-import(io_widget,
        [get_state/1, insert_str/2, set_prompt/2, set_state/2,
         set_title/2, set_handler/2, update_state/3]).

-export([start/0, test/0, connect/5]).


start() ->
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "joe" ).

test() ->
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "joe" ),
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "jane" ),
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "jim" ),
    connect("localhost" , 2223, "AsDT67aQ" , "general" , "sue" ).


connect(Host, Port, HostPsw, Group, Nick) ->
    spawn(fun() -> handler(Host, Port, HostPsw, Group, Nick) end).

handler(Host, Port, HostPsw, Group, Nick) ->
    process_flag(trap_exit, true),
    Widget = io_widget:start(self()),
    set_title(Widget, Nick),
    set_state(Widget, Nick),
    set_prompt(Widget, [Nick, " > " ]),
    set_handler(Widget, fun parse_command/1),
    start_connector(Host, Port, HostPsw),
    disconnected(Widget, Group, Nick).

disconnected(Widget, Group, Nick) ->
    receive
        {connected, MM} ->
            insert_str(Widget, "connected to server\\nsending data\\n" ),
            MM ! {login, Group, Nick},
            wait_login_response(Widget, MM);
        {Widget, destroyed} ->
            exit(died);
        {status, S} ->
            insert_str(Widget, to_str(S)),
            disconnected(Widget, Group, Nick);
        Other ->
            io:format("chat_client disconnected unexpected:\`~p\`~n" ,[Other])
            disconnected(Widget, Group, Nick)
    end.

```

```

wait_login_response(Widget, MM) ->
    receive
        {MM, ack} ->
            active(Widget, MM);
        Other ->
            io:format("chat_client login unexpected:\~p\~n" ,[Other]),
            wait_login_response(Widget, MM)
    end.

active(Widget, MM) ->
    receive
        {Widget, Nick, Str} ->
            MM ! {relay, Nick, Str},
            active(Widget, MM);
        {MM,{msg,From,Pid,Str}} ->
            insert_str(Widget, [From,"@",pid_to_list(Pid)," ",Str, "\n"],
            active(Widget, MM));
        {'EXIT',Widget,windowDestroyed} ->
            MM ! close;
        {close, MM} ->
            exit(serverDied);
    end.

Other ->
    io:format("chat_client active unexpected:\~p\~n" ,[Other]),
    active(Widget, MM)
end.

start_connector(Host, Port, Pwd) ->
    S = self(),
    spawn_link(fun() -> try_to_connect(S, Host, Port, Pwd) end).
try_to_connect(Parent, Host, Port, Pwd) ->
    %% Parent is the Pid of the process that spawned this process
    case lib_chan:connect(Host, Port, chat, Pwd, []) of
        {error, _Why} ->
            Parent ! {status, {cannot, connect, Host, Port}},
            sleep(2000),
            try_to_connect(Parent, Host, Port, Pwd);
        {ok, MM} ->
            lib_chan_mm:controller(MM, Parent),
            Parent ! {connected, MM},
            exit(connectorFinished)
    end.

sleep(T) ->
    receive
        after T -> true
    end.

```

```

end.

to_str(Term) ->
    io_lib:format("\~p\~n" ,[Term]) .

parse_command(Str) -> skip_to_gt(Str).

skip_to_gt(">" ++ T)      -> T;
skip_to_gt([_ | T])        -> skip_to_gt(T);
skip_to_gt([])             -> exit("no >").

```

Конфигурация lib_chan

[Скачать socket_dist/chat.conf](#)

```

{port, 2223}.
{service, chat, password,"AsDT67aQ",mfa,mod_chat_controller,start,[]}.

```

Чат-контроллер

[Скачать socket_dist/mod_chat_controller.erl](#)

```

-module(mod_chat_controller).
-export([start/3]).
-import(lib_chan_mm, [send/2]).

start(MM, _, _) ->
    process_flag(trap_exit, true),
    io:format("mod_chat_controller off we go ... \~p\~n" ,[MM]),
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, Msg} ->
            chat_server ! {mm, MM, Msg},
            loop(MM);
        {'EXIT', MM, _Why} ->
            chat_server ! {mm_closed, MM};
        Other ->
            io:format("mod_chat_controller unexpected message =\~p (MM=\~p)
[Other, MM]),
            loop(MM)
    end.

```

Чат-сервер

Скачать [socket_dist/chat_server.erl](#)

```
-module(chat_server).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [delete/2, foreach/2, map/2, member/2,reverse/2])�.

-compile(export_all).

start() ->
    start_server(),
    lib_chan:start_server("chat.conf" ).

start_server() ->
    register(chat_server,
    spawn(fun() ->
        process_flag(trap_exit, true),
        Val = (catch server_loop([])),
        io:format("Server terminated with:\`~p\`~n" ,[Val])
    end)).

server_loop(L) ->
    receive
        {mm, Channel, {login, Group, Nick}} ->
            case lookup(Group, L) of
                {ok, Pid} ->
                    Pid ! {login, Channel, Nick},
                    server_loop(L);
                error ->
                    Pid = spawn_link(fun() ->
                        chat_group:start(Channel, Nick)
                    end),
                    server_loop([{Group,Pid}|L])
            end;
        {mm_closed, _} ->
            server_loop(L);
        {'EXIT', Pid, allGone} ->
            L1 = remove_group(Pid, L),
            server_loop(L1);
        Msg ->
            io:format("Server received Msg=\`~p\`~n" ,[Msg]),
            server_loop(L)
    end.

```

```

lookup(G, [{G,Pid}|_])  -> {ok, Pid};
lookup(G, [_|T])         -> lookup(G, T);
lookup(_, [])             -> error.

remove_group(Pid, [{G,Pid}|T])  -> io:format("\~p removed\~n" ,[G]), T;
remove_group(Pid, [H|T])        -> [H|remove_group(Pid, T)];
remove_group(_, [])            -> [].

```

Чат-группы

[Скачать socket_dist/chat_group.erl](#)

```

-module(chat_group).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [foreach/2, reverse/2]).


-export([start/2]).


start(C, Nick) ->
    process_flag(trap_exit, true),
    controller(C, self()),
    C ! ack,
    self() ! {C, {relay, Nick, "I'm starting the group" }},
    group_controller([{C,Nick}]).


delete(Pid, [{Pid,Nick}|T], L)  -> {Nick, reverse(T, L)};
delete(Pid, [H|T], L)           -> delete(Pid, T, [H|L]);
delete(_, [], L)               -> {"?????", L}.


group_controller([]) ->
    exit(allGone);
group_controller(L) ->
    receive
        {C, {relay, Nick, Str}} ->
            foreach(fun({Pid,_}) -> Pid ! {msg, Nick, C, Str} end, L),
            group_controller(L);
        {login, C, Nick} ->
            controller(C, self()),
            C ! ack,
            self() ! {C, {relay, Nick, "I'm joining the group" }},
            group_controller([{C,Nick}|L]);
        {close,C} ->
            {Nick, L1} = delete(C, L, []),
            self() ! {C, {relay, Nick, "I'm leaving the group" }},
            group_controller(L1);
    end.

```

```

Any ->
    io:format("group controller received Msg=~p~n" , [Any]),
    group_controller(L)
end.

```

Виджет ввода-вывода

Скачать [socket_dist/io_widget.erl](#)

```

-module(io_widget).

-export([get_state/1,
         start/1, test/0,
         set_handler/2,
         set_prompt/2,
         set_state/2,
         set_title/2, insert_str/2, update_state/3]).

start(Pid) ->
    gs:start(),
    spawn_link(fun() -> widget(Pid) end).

get_state(Pid)      -> rpc(Pid, get_state).
set_title(Pid, Str) -> Pid ! {title, Str}.
set_handler(Pid, Fun) -> Pid ! {handler, Fun}.
set_prompt(Pid, Str) -> Pid ! {prompt, Str}.
set_state(Pid, State) -> Pid ! {state, State}.
insert_str(Pid, Str) -> Pid ! {insert, Str}.
update_state(Pid, N, X) -> Pid ! {updateState, N, X}.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, R} ->
            R
    end.

widget(Pid) ->
    Size = [{width,500},{height,200}],
    Win = gs:window(gs:start(),
                    [{map,true},{configure,true},{title,"window"}|Size]),
    gs:frame(packer, Win,[{packer_x, [{stretch,1,500}]},
                           {packer_y, [{stretch,10,120,100},
                                       {stretch,1,15,15}]}]),
    gs:create(editor,editor,packer, [{pack_x,1},{pack_y,1},{vscroll,right}])

```

```

gs:create(entry, entry, packer, [{pack_x,1},{pack_y,2},{keypress,true}])
gs:config(packer, Size),
Prompt = " > ",
State = nil,
gs:config(entry, {insert,{0,Prompt}}),
loop(Win, Pid, Prompt, State, fun parse/1).

loop(Win, Pid, Prompt, State, Parse) ->
receive
    {From, get_state} ->
        From ! {self(), State},
        loop(Win, Pid, Prompt, State, Parse);
    {handler, Fun} ->
        loop(Win, Pid, Prompt, State, Fun);
    {prompt, Str} ->
        %% this clobbers the line being input ...
        %% this could be fixed - hint
        gs:config(entry, {delete,{0,last}}),
        gs:config(entry, {insert,{0,Str}}),
        loop(Win, Pid, Str, State, Parse);
    {state, S} ->
        loop(Win, Pid, Prompt, S, Parse);
    {title, Str} ->
        gs:config(Win, [{title, Str}]),
        loop(Win, Pid, Prompt, State, Parse);
    {insert, Str} ->
        gs:config(editor, {insert,['end',Str]}),
        scroll_to_show_last_line(),
        loop(Win, Pid, Prompt, State, Parse);
    {updateState, N, X} ->
        io:format("setelemtn N=~p X=~p Satte=~p~n" ,[N,X,State])
        State1 = setelement(N, State, X),
        loop(Win, Pid, Prompt, State1, Parse);
    {gs,_,destroy,_,_} ->
        io:format("Destroyed~n" ,[]),
        exit(windowDestroyed);
    {gs, entry, keypress, _, ['Return'|_] } ->
        Text = gs:read(entry, text),
        %% io:format("Read:~p~n",[Text]),
        gs:config(entry, {delete,{0,last}}),
        gs:config(entry, {insert,{0,Prompt}}),
        try Parse(Text) of
            Term ->
                Pid ! {self(), State, Term}
        catch

```

```

        -:_ ->
            self() ! {insert, "** bad input**\\n** /h for help\\n"}.
        end,
        loop(Win, Pid, Prompt, State, Parse);
{gs,_,configure,[],[W,H,_,_]} ->
    gs:config(packer, [{width,W},{height,H}]),
    loop(Win, Pid, Prompt, State, Parse);
{gs, entry,keypress,_,_} ->
    loop(Win, Pid, Prompt, State, Parse);
Any ->
    io:format("Discarded:\~p\~n" ,[Any]),
    loop(Win, Pid, Prompt, State, Parse)
end.

scroll_to_show_last_line() ->
    Size      = gs:read(editor, size),
    Height     = gs:read(editor, height),
    CharHeight = gs:read(editor, char_height),
    TopRow     = Size - Height/CharHeight,
    if TopRow > 0 -> gs:config(editor, {vscrollpos, TopRow});
        true       -> gs:config(editor, {vscrollpos, 0})
    end.

test() ->
    spawn(fun() -> test1() end).

test1() ->
    W = io_widget:start(self()),
    io_widget:set_title(W, "Test window" ),
    loop(W).

loop(W) ->
    receive
        {W, {str, Str}} ->
            Str1 = Str ++ "\n",
            io_widget:insert_str(W, Str1),
            loop(W)
    end.

parse(Str) ->
    {str, Str}.

```

11.7 Упражнения

- улучшите графический виджет, добавив боковую панель для перечисления имён участников группы
 - добавьте код для показа имён всех участников группы
 - добавьте код для перечисления всех групп
 - добавьте личные сообщения
 - добавьте такой код, чтобы контроллер группы работал не на серверной машине, а на машине первого пользователя, который подключился к данной группе
 - Посмотрите внимательно на диаграмму последовательности сообщений (Рис.11. 2), чтобы убедиться, что вы понимаете её и проверьте, что вы можете указать все сообщения из диаграммы в программном коде
 - нарисуйте свою собственную диаграмму последовательности сообщений, чтобы показать, как решается проблема в фазе логина (в оригинале «фаза логина проблемы»)
-

(1) Это облегчает нам жизнь и позволяет сосредоточиться на приложении вместо низкоуровневых деталей протокола.

(2) **MM** означает middle man — посредник. Это процесс, который используется для связи с сервером.

(3) **Nick** — это прозвище пользователя

Глава 12. Интерфейсы с другими программами

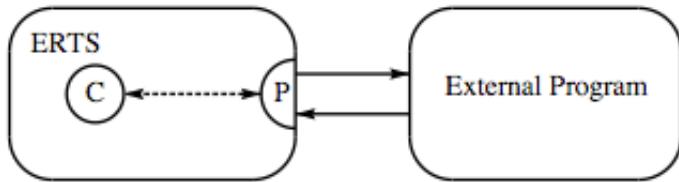
Предположим мы хотим связать программу на Эрланге с программой на С или Питоне или-же запустить из Эрланга командную консоль . Чтобы это сделать, нам надо запустить внешнюю программу в отдельном процессе операционной системы, отдельно от системы исполнения приложений (runtime) Эрланга и обмениваться с этой программой сообщениями через байт-ориентированный коммуникационный канал. Со стороны Эрланга, часть отвечающую за такую связь, называют порт. Процесс, который создает этот порт, называют процессом подключенным к этому порту. Подключенный процесс имеет особое значение для Эрланга: все сообщения к внешней программе помечаются PID-ом подключенного процесса, и все сообщения от внешней программы посылаются ему-же.

Взаимоотношения между подключенным процессом **C** портом **P** и внешним процессом операционной системы, показаны на Рисунке 12.1.

ЭСИП (ERTS) — Система исполнения приложений Эрланга

C — Процесс Эрланга подключенный к порту

P – порт.



ERTS = Erlang runtime system

C = An Erlang process that is connected to the port

P = A port

Рисунок 12.1: Коммуникации с портом.

С точки зрения программиста порт ведет себя аналогично Эрланговскому процессу.

Вы можете посыпать ему сообщения можете зарегистрировать его (аналогично процессу) и так далее. Если внешняя программа обвалится (упадет), то сигнал exit будет послан подсоединенному к порту процессу, а если умрет подсоединеный процесс, то и внешняя программа будет убита killed.

Возможно, вы удивитесь, почему все сделано именно так? Многие языки программирования позволяют просто подлинковывать программы на других языках программирования в свои исполняемые приложения. В Эрланге, мы этого не позволяем из соображений безопасности(1) (исключение составляют подключаемые драйверы, которые мы обсудим далее в этой главе). Если бы мы просто подлинковали внешнюю программу в исполняемую среду Эрланга, то тогда ошибка в этой внешней программе могла бы легко повалить всю систему Эрланга. По этой причине, все другие языки должны исполняться снаружи системы Эрланга, в отдельном исполняемом процессе операционной системы. Система исполнения Эрланга и внешние процессы взаимодействуют (обмениваются информацией) через поток байтов.

12.1 Порты

Чтобы создать порт мы даем следующую команду:

```
Port = open_port(PortName, PortSettings)
```

Она возвращает нам порт. Следующие типы сообщений могут быть посланы порту(2) (`PidC` означает `PID` подключенного к порту процесса):

`Port!{PidC, {command, Data}}` – посыпает данные `Data` (байтовый список) порту.

`Port!{PidC, {connect, Pid1}}` – изменяет подключенный к порту процесс (с `PID`-ом

`PidC`) на процесс с PID-ом `Pid1`.

`Port!{PidC,close}` – закрывает данный порт.

Подключенный к порту процесс может получать сообщения от внешней программы порта следующим стандартным образом:

```
receive
  {Port, {data, Data} ->
  ...обработка данных, полученных от внешней программы...
```

В следующих разделах мы рассмотрим интерфейс Эрланга с очень простой С-программой, которая намеренно будет очень короткой, чтобы не отвлекать читателя от рассмотрения деталей интерфейса.

Замечание: Следующий пример намеренно выбран очень простым, чтобы можно было выделить в рассмотрении механизмы и протоколы общения с портом. Кодирование и декодирование сложных структур данных для обмена через протоколы порта — это сложная проблема, которую мы здесь не в состоянии решить. В конце этой главы мы дадим ссылки на некоторые библиотеки, которые могут быть использованы для построения интерфейсов к другим языкам программирования.

12.2 Интерфейс к внешней программе на С

Сначала мы посмотрим на нашу простейшую С-программу, выбранную для нашего примера:

Загрузить `ports/example1.c`

```
int twice(int x) {
    return 2*x;
}

int sum(int x, int y) {
    return x+y;
}
```

Нашей конечной целью будет вызов этих С-функций из Эрланга. Мы хотим иметь возможность написать (на Эрланге):

```
X1 = example1:twice(23),
Y1 = example1:sum(45, 32),
```

То есть с точки зрения пользователя, `example1` должен выглядеть, как обычный модуль Эрланга и, следовательно, все детали его взаимодействия (интерфейса) с программой на С, должны быть скрыты внутри модуля `example1.erl`.

Нашему интерфейсу потребуется основная С-программа `main`, которая будет декодировать данные присланные от Эрланг-программы. В нашем примере мы, в начале, определим протокол между портом и внешней С-программой. Он будет крайне простым и мы покажем как его реализовать на Эрланге и на С. Протокол будет следующим:

- Все пакеты начинаются с двух-байтового кода их длины `Len` за которым будут следовать `Len` байтов данных.
- Чтобы вызвать С-функцию `twice(N)` Эрланг-программа должна как-то закодировать такой вызов, согласованным образом (по договоренности). Мы будем полагать что это будет 2-х байтоваая последовательность `[1,N]`, где 1 обозначает вызов вызов С-функции `twice`, а N — является ее одно-байтовым аргументом.
- Для вызова С-функции `sum(N,M)` мы, аналогично, будем использовать байтовую последовательность `[2,N,M]`.
- Предполагается, что возвращаемые значения будут длиной в 1 байт.

И программа на С, и программа на Эрланге должны следовать этому протоколу. Давайте, например, рассмотрим по шагам, что должно произойти, если Эрланг-программа захочет вычислить `sum(45,32)`:

1. Порт посыпает внешней программе байтовую последовательность `0,3,2,45,32`. Первые ее два байта представляют длину посыпаемого пакета `3`; следующий код `2` — означает вызов внешней функции `sum`; а 45 и 32 — это ее одно-байтовые аргументы.
2. Внешняя программа читает эти 5 байт из своего стандартного входного потока `input`, вызывает функцию `sum` с переданными аргументами, и, потом, записывает байтовую последовательность `0,1,77` в свой стандартный выходной поток `output`. Первые два байта представляют из себя длину следующего пакета. А за ними следует однобайтовый результат работы функции `sum(45,32)` — `77`.

Теперь нам надо написать программы на обоих сторонах интерфейса, которые строго следуют этому протоколу. Мы начнем с программы на С.

С-программа

Внешняя С-программа будет состоять из следующих трех файлов:

`example1.c` — он будет содержать функции, которые мы хотим вызывать.

`example1_driver.c` – здесь будет реализован байтовый протокол и вызываться нужные функции из `example1.c`.

`erl_comm.c` – здесь будут реализованы нужные процедуры чтения и записи буферов памяти.

example1_driver.c

Загрузить [ports/example1_driver.c](#)

```
#include <stdio.h>
typedef unsigned char byte;

int read_cmd(byte *buff);
int write_cmd(byte *buff, int len);

int main() {
    int fn, arg1, arg2, result;
    byte buff[100];

    while (read_cmd(buff) > 0) {
        fn = buff[0];
        if (fn == 1) {
            arg1 = buff[1];
            result = twice(arg1);
        } else if (fn == 2) {
            arg1 = buff[1];
            arg2 = buff[2];
            /* debug -- you can print to stderr to debug
               fprintf(stderr,"calling sum %i %i\\n",arg1,arg2); */
            result = sum(arg1, arg2);
        }
        buff[0] = result;
        write_cmd(buff, 1);
    }
}
```

Этот код работает в бесконечном цикле, читая команды из стандартного входного потока (`input`), вызывает нужные функции и записывает результаты в стандартный выходной поток (`output`).

Если вы хотите использовать отладочную печать в С-программе, то вы должны направить ее вывод в `stderr`. Пример отладочной печати приведен в закомментированном участке кода программы.

(1) Исключением из этого правила является использование связанных драйверов, что мы обсудим позже в этой главе.

(2) Во всех этих сообщениях, `PidC` является *PID*-ом подключенного процесса.

erl_comm.c

Этот код предназначен для обработки пакетов с заголовком из 2-х байт, который будет соответствовать опции `{packet,2}` для программы порта драйвера (см. ниже).

Загрузить [ports/erl_comm.c](#)

```
/* erl_comm.c */
#include <unistd.h>

typedef unsigned char byte;

int read_cmd(byte *buf);
int write_cmd(byte *buf, int len);
int read_exact(byte *buf, int len);
int write_exact(byte *buf, int len);

int read_cmd(byte *buf)
{
    int len;

    if (read_exact(buf, 2) != 2)
        return(-1);
    len = (buf[0] << 8) | buf[1];
    return read_exact(buf, len);
}

int write_cmd(byte *buf, int len)
{
    byte li;

    li = (len >> 8) & 0xff;
    write_exact(&li, 1);

    li = len & 0xff;
    write_exact(&li, 1);

    return write_exact(buf, len);
}

int read_exact(byte *buf, int len)
{
```

```

int i, got=0;

do {
    if ((i = read(0, buf+got, len-got)) <= 0)
        return(i);
    got += i;
} while (got<len);

return(len);
}

int write_exact(byte *buf, int len)
{
    int i, wrote = 0;

    do {
        if ((i = write(1, buf+wrote, len-wrote)) <= 0)
            return (i);
        wrote += i;
    } while (wrote<len);

    return (len);
}

```

Программа на Эрланге

Драйвер порта со стороны Эрланга обеспечивается следующим модулем:

[Загрузить ports/example1.erl](#)

```

-module(example1).
-export([start/0, stop/0]).
-export([twice/1, sum/2]).


start() ->
    spawn(fun() ->
        register(example1, self()),
        process_flag(trap_exit, true),
        Port = open_port({spawn, "./example1" }, [{packet, 2}]),
        loop(Port)
    end).

stop() ->
    example1 ! Stop.

```

```

twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).

call_port(Msg) ->
    example1 ! {call, self(), Msg},
    receive
        {example1, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {example1, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit({port_terminated,Reason})
    end.

encode({twice, X}) -> [1, X];
encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.

```

Порт открывается следующей командой:

```
Port = open_port({spawn, "./example1"}, [{packet, 2}])
```

Опция `{packet, 2}` говорит системе автоматически добавлять к пакетам, адресованным удаленной программе, 2-х байтовый заголовок длины этого пакета. Поэтому, если мы пошлем сообщение `{PidC, {command, [2,45,32]}}` порту, то драйвер этого порта добавит 2-х байтовую длину в заголовок пакета и пошлет последовательность `0,3,2,45,32` внешней программе.

При приеме данных, порт также будет полагать, что каждый входящий пакет

предваряется 2-х байтовым заголовком и будет удалять его байты до того как передать данные подключенному к порту процессу Эрланга.

Давайте соберем наши программы. Мы используем для этого следующий make-файл для их построения. Команда make example1 собирает внешнюю программу, которая (ее имя) используется как аргумент в Эрланговской функции `open_port`. Заметьте, что данный make-файл также включает в себя код для построения прилинкованного драйвера, который будет рассмотрен далее в этой главе.

Make-файл

[Загрузить ports/Makefile](#)

```
.SUFFIXES: .erl .beam .yrl

.erl.beam:
    erlc -W $<

MODS = example1 example1_lid

all: ${MODS:%=%.beam} example1 example1_drv.so

example1: example1.c erl_comm.c example1_driver.c
    gcc -o example1 example1.c erl_comm.c example1_driver.c

example1_drv.so: example1_lid.c example1.c
    gcc -o example1_drv.so -fpic -shared example1.c example1_lid.c

clean:
    rm example1 example1_drv.so *.beam
```

Запуск программы

Теперь мы можем запустить нашу программу:

```
1> example1:start().
<0.32.0>
2> example1:sum(45, 32).
77
4> example1:twice(10).
20
...
...
```

На чем мы и завершим наш первый пример.

Но, до того как мы перейдем к следующему разделу, мы должны отметить следующее:

- В данной программе не делается попыток унификации представления С и Эрланга о том, что такое есть целое число. Мы просто полагаем, что целые в Эрланге и С у нас это просто однобайтовые числа и игнорируем все возможные проблемы точности представления и знаков. В реальных приложениях, нам придется гораздо серьезнее задуматься над типами и их представлениями для передаваемых данных. Это может быть не простым вопросом, поскольку Эрланг свободно манипулирует целыми числами произвольной размерности, в то время как такие языки как С имеют различные фиксированные представления для целых определенных размерностей и так далее.
- Мы не можем просто запустить Эрланг-функции, без предварительного запуска драйвера, который отвечает за интерфейс (то есть, какая-то программа должна до этого выполнить `example1:start()`, прежде чем мы сможем запустить нашу программу). Нам бы, конечно, хотелось, чтобы это происходило автоматически, во время старта нашей системы. Это вполне возможно, но для этого требуются некоторые знания на тему того, как система стартует и останавливается. Мы рассмотрим эти вопросы позже в разделе 18.7 Приложения (Эрланга).

12.3 open_port

В предыдущем разделе мы использовали функцию `open_port` без подробного рассказа каковы бывают ее аргументы и что они, при этом, делают. Мы видели только использование `open_port` с аргументом `{packet, 2}`, который добавляет и убирает 2-х байтовый заголовок — длину пакета для данных, пересылаемых между Эрлангом и внешней программой. На самом деле у `open_port` может быть довольно много других аргументов.

Некоторые, из наиболее используемых, из них приведены далее:

```
@spec open_port(PortName, [0pt]) -> Port
```

`PortName` может иметь следующие виды:

```
{spawn, Command}
```

Запускает внешнюю программу. `Command` — имя этой внешней программы, которая запускается вне рабочего пространства Эрланга, если только не найдется прилинкованный драйвер с именем `Command`.

```
{fd, In, Out}
```

Позволяет Эрланговскому процессу получить доступ к любому открытому файловому дескриптору, который использует Эрланг. Файловые дескрипторы `In` может быть использован для стандартного ввода, а файловый дескриптор `Out` — для стандартного вывода (см пример подключения стандартного ввода и вывода по ссылке: <http://ftp.sunet.se/pub/lang/erlang/examples/examples-2.0.html> (Прим. редактора: Оригинальный линк из книги не доступен.)

Опции `Opt` могут быть следующими:

`{packet, N}`

Пакеты будут предваряться N-байтовым (N=1,2,4) счетчиком байт в пакете данных.

`stream`

Сообщения пересылаются без подсчета их длины. Приложение должно само уметь обрабатывать такие пакеты данных.

`{line, Max}`

Обмен сообщениями на основе принципа по одному-в-строке. Если строка более чем `Max` байт, то она разбивается после `Max` байт на следующую строку (и так далее).

`{cd, Dir}`

Действует только для параметра `{spawn, Command}`. Внешняя программа запускается в директории `Dir`.

`{env, Env}`

Действует только для параметра `{spawn, Command}`. Переменные окружения для внешней программы расширяются переменными из списка `Env`, состоящего из пар вида `{VarName, Value}` (`{ИмяПеременной, ЕеЗначение}`), где `VarName` и `Value` — это строки.

Это не полный список аргументов для функции `open_port`. Их полное описание можно найти в документации для модуля `erlang`.

12.4 Подключаемые драйверы

Иногда возникает потребность, чтобы программа написанная на другом языке

работала внутри системы исполнения приложений Эрланга. В этом случае, программа пишется как разделяемая библиотека, которая динамически подлинковывается к системе исполнения Эрланга. Подключаемые драйверы выглядят для программиста так же, как и программы порта и подчиняются точно тем-же протоколам, что и они.

Создание подключаемых драйверов — это самый эффективный путь взаимодействия с кодом на другом языке из Эрланга, но он, также, и самый опасный. Любая фатальная ошибка в подключенном драйвере повалит всю систему исполнения приложений Эрланга со всеми запущенными в ней процессами. По этой причине, использование подключаемых драйверов не рекомендуется; и их следует использовать только тогда, когда все возможные альтернативы не подходят.

Чтобы проиллюстрировать этот подход, мы превратим использованную нами ранее программу в подключаемый драйвер. Чтобы это сделать нам потребуется три файла:

`example1_lid.erl` – это Эрланг сервер.

`example1.c` – содержит С-функции, которые мы хотим использовать. Ничем не отличается от использованного нами ранее.

`example1_lid.c` – это С-программа, которая вызывает С-функции из `example1.c`

Код Эрланга, поддерживающий такой интерфейс приведен далее:

Загрузить [ports/example1_lid.erl](#)

```
-module(example1_lid).
-export([start/0, stop/0]).
-export([twice/1, sum/2]).

start() ->
    start("example1_drv" ).
start(SharedLib) ->
    case erl_ddll:load_driver(.. , SharedLib) of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    spawn(fun() -> init(SharedLib) end).

init(SharedLib) ->
    register(example1_lid, self()),
    Port = open_port({spawn, SharedLib}, []),
    loop(Port).
```

```

stop() ->
    example1_lid ! stop.

twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).

call_port(Msg) ->
    example1_lid ! {call, self(), Msg},
    receive
        {example1_lid, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {example1_lid, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            io:format("\~p \~n" , [Reason]),
            exit(port_terminated)
    end.

encode({twice, X}) -> [1, X];
encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.

```

Если мы сравним эту программу с ее предыдущей версией для интерфейса порта, мы увидим, что они практически идентичны.

Программа подключаемого драйвера состоит большей частью из кода работающего с элементами его структуры `driver`. Команда `make example1_drv.so` для `make`-файла, приведенного нами ранее, позволяет построить нужную разделяемую библиотеку данного драйвера.

ports/example1_lid.c

```
/* example1_lid.c */

#include <stdio.h>
#include "erl_driver.h"

typedef struct {
    ErlDrvPort port;
} example_data;

static ErlDrvData example_drv_start(ErlDrvPort port, char *buff)
{
    example_data* d = (example_data*)driver_alloc(sizeof(example_data));
    d->port = port;
    return (ErlDrvData)d;
}

static void example_drv_stop(ErlDrvData handle)
{
    driver_free((char*)handle);
}

static void example_drv_output(ErlDrvData handle, char *buff, int bufflen)
{
    example_data* d = (example_data*)handle;
    char fn = buff[0], arg = buff[1], res;
    if (fn == 1) {
        res = twice(arg);
    } else if (fn == 2) {
        res = sum(buff[1], buff[2]);
    }
    driver_output(d->port, &res, 1);
}

ErlDrvEntry example_driver_entry = {
    NULL,           /* F_PTR init, N/A */
    example_drv_start, /* L_PTR start, called when port is opened */
    example_drv_stop, /* F_PTR stop, called when port is closed */
    example_drv_output, /* F_PTR output, called when erlang has sent
                         data to the port */
    NULL,           /* F_PTR ready_input,
                     called when input descriptor ready to read*/
    NULL,           /* F_PTR ready_output,
                     called when output descriptor ready to write */
}
```

```

"example1_drv",      /* char *driver_name, the argument to open_port */
NULL,                /* F_PTR finish, called when unloaded */
NULL,                /* F_PTR control, port_command callback */
NULL,                /* F_PTR timeout, reserved */
NULL                /* F_PTR outputv, reserved */

};

DRIVER_INIT(example_drv) /* must match name in driver_entry */
{
    return &example_driver_entry;
}

```

Вот результаты работы этих программ:

```

1> **c(example1_lid).
{ok,example1_lid}
2> **example1_lid:start().
<0.41.0>
3> **example1_lid:twice(50).
100
4> **example1_lid:sum(10, 20).
30

```

12.5 Примечания

В этой главе мы рассмотрели как использовать порты для взаимодействия из Эрланга с внешними программами. В дополнение к протоколу порта, можно использовать еще несколько BIF для работы с ними. Все это описано в документации к модулю erlang .

Но сейчас у вас, возможно, возникает вопрос, а как передавать сложные структуры данных между Эрлангом и внешними программами? Как пересыпать строки, кортежи и так далее? К сожалению, не существует простого ответа на этот вопрос и порт предоставляет только низкоуровневый протокол обмена последовательностями байтов между Эрлангом и внешним миром. Между прочим, точно такая же проблема существует и для socket-обмена данными. Сокеты обеспечивают только потоки байтов между двумя приложениями, а как их интерпретировать — полностью перекладывается на сами эти приложения.

Тем не менее, существует несколько библиотек, включенных в дистрибутивы Эрланга, которые облегчают проблемы общения Эрланговских программ с внешними программами. Они включают в себя следующие примеры:

http://www.erlang.org/doc/apps/erl_interface/erl_interface.pdf

Интерфейс Erl (ei) – это набор С-функций и макросов для кодирования и декодирования Эрланговских форматов. На стороне Эрланга используется функция `term_to_binary` для представления Эрланговского терма (объекта) в виде байтовой последовательности. А на стороне С-программы, указанные функции из ei могут быть использованы для распаковки этих бинарных данных. Кроме того, ei можно использовать и для создания бинарных данных, которые на стороне Эрланга распаковываются с помощью `binary_to_term`.

<http://www.erlang.org/doc/apps/ic/ic.pdf>

IDL компилятор Эрланга (ic). Приложение ic – это реализация на Эрланге OMG IDL компилятора.

<http://www.erlang.org/doc/apps/jinterface/jinterface.pdf>

Jinterface – это набор средств обеспечивающих взаимодействие Джавы и Эрланга. Он обеспечивает полное отображение типов Эрланга в объекты Джавы, кодирование и декодирование Эрланговских термов, связь с процессами Эрланга и так далее, включая большой набор дополнительных средство и возможностей.

Глава 13. Работа с файлами

В этой главе мы рассмотрим некоторые общие функции для манипулирования файлами. В стандартном релизе Эрланга есть множество функций для работы с файлами. Мы сосредоточимся на той малой их части, которую я использую в большинстве моих программ. Мы также рассмотрим некоторые примеры техники, которую я использую для написания эффективного кода обработки файлов. В дополнение к этому, я кратко упомяну некоторые реже используемые файловые операции, чтобы вы знали, что они есть. А если вы захотите узнать больше подробностей – обращайтесь к мануалам.

Мы сосредоточимся на следующих областях:

- организация библиотек;
 - разные способы чтения файлов;
 - разные способы записи файлов;
 - работа с директориями;
 - поиск информации о файлах.
-

13.1 Организация библиотек

Функции для манипулирования файлами организованы в четыре модуля:

`file` – функции для открытия, закрытия, чтения и записи файлов; просмотра директорий и т. д. Краткая сводка часто используемых функций приведена на Таблице 13.1. За полными подробностями обращайтесь к руководству по модулю `file`.

`filename` – этот модуль содержит функции, которые манипулируют именами файлов платформонезависимым образом, так, что вы можете выполнять один и тот же код на разных операционных системах.

`filelib` – этот модуль – дополнение к `file`, который содержит ряд вспомогательных функций для просмотра файлов, проверки типов файлов и т.п. Большинство из них написаны, используя функции из `file`.

`io` – этот модуль содержит функции, которые работают с открытыми файлами. Он содержит функции для парсинга (разбора) данных в файле и записи форматированных данных в файл.

Функция	Описание
<code>change_group</code>	Сменить группу у файла
<code>change_owner</code>	Сменить владельца у файла
<code>change_time</code>	Сменить время модификации или последнего доступа у файла
<code>close</code>	Закрыть файл
<code>consult</code>	Прочитать термы Эрланга из файла
<code>copy</code>	Копировать содержимое файла
<code>del_dir</code>	Удалить директорию
<code>delete</code>	Удалить файл
<code>eval</code>	Выполнить выражения Эрланга из файла
<code>format_error</code>	Вернуть строку с описанием причины ошибки
<code>get_cwd</code>	Получить текущую директорию
<code>list_dir</code>	Вывести список файлов в директории
<code>make_dir</code>	Создать директорию
<code>make_link</code>	Создать hard ссылку на файл
<code>make_symlink</code>	Создать soft (символическую) ссылку на файл
<code>open</code>	Открыть файл

<code>position</code>	Установить позицию в файле
<code>pread</code>	Читать из файла с указанной позиции
<code>pwrite</code>	Записать в файл с указанной позиции
<code>read</code>	Читать из файла
<code>read_file</code>	Прочитать весь файл целиком
<code>read_file_info</code>	Получить информацию о файле
<code>read_link</code>	Посмотреть — куда указывает ссылка
<code>read_link_info</code>	Получить информацию о ссылке или файле
<code>rename</code>	Переименовать файл
<code>script</code>	Выполнить и вернуть значение выражений Эрланга из файла
<code>set_cwd</code>	Установить текущую директорию
<code>sync</code>	Синхронизировать состояние файла в памяти и на физическом носителе
<code>truncate</code>	Обрезать файл
<code>write</code>	Записать в файл
<code>write_file</code>	Записать весь файл целиком
<code>write_file_info</code>	Сменить информацию о файле

Таблица 13.1: Сводка файловых операций (модуль `file`)

Разные способы чтения файлов

Давайте глянем – что у нас есть, когда дело доходит до чтения файлов. Мы начнём с написания пяти маленьких программок, которые открывают файл и берут оттуда данные несколькими разными способами.

Содержимое файла — это просто последовательность байт. Что они значат — зависит от интерпретации этих байтов.

Чтобы продемонстрировать это мы будем использовать один и тот же файл для всех наших примеров. Вообще-то, он содержит последовательность термов Эрланга. В зависимости от того, как мы открываем и читаем файл, мы можем интерпретировать содержимое как последовательность термов Эрланга, как последовательность текстовых строк или как куски бессмысленных и беспощадных бинарных данных.

Загрузить `data1.dat`

```
{person, "joe" , "armstrong" ,
```

```
[{occupation, programmer},  
 {favoriteLanguage, erlang}]}].  
{cat, {name, "zorro" }},  
 {owner, "joe" }}].
```

А теперь мы прочитаем части этого файла разными способами.

Чтение всех термов из файла

`data1.dat` содержит последовательность термов Эрланга. Мы можем прочитать их все, используя функцию `file:consult` следующим образом:

```
1> file:consult("data1.dat").  
{ok,[{person,"joe",  
"armstrong",  
[{occupation,programmer},{favoriteLanguage,erlang}]],  
{cat,{name,"zorro"},{owner,"joe"}}]}
```

`file:consult(File)` полагает, что `File` содержит последовательность термов Эрланга. Она возвращает `{ok, [Term]}`, если может прочитать все термы из файла. В противном случае она возвращает `{error, Reason}`.

Чтение термов по одному за раз

Если мы хотим прочитать термы из файла по одному за раз, то мы открываем файл функцией `file:open`, а затем читаем отдельные термы функцией `io:read` до тех пор, пока не дойдём до конца файла. Затем мы закрываем файл функцией `file:close`.

Вот сеанс в оболочке Эрланга, который показывает что происходит, когда мы читаем термы из файла по одному за раз:

```
1> {ok, S} = file:open("data1.dat", read).  
{ok,<0.36.0>}  
2> io:read(S, '').  
{ok,{person,"joe",  
"armstrong",  
[{occupation,programmer},{favoriteLanguage,erlang}]}  
3> io:read(S, '').  
{ok,{cat,{name,"zorro"},{owner,"joe"}}}  
4> io:read(S, '').  
eof  
5> file:close(S)
```

Функции, которые мы здесь использовали, следующие:

```
@spec file:open(File, read) => {ok, IoDevice} | {error, Why}
```

Пытается открыть файл `File` для чтения. Возвращает `{ok, IoDevice}` в случае успеха, либо `{error, Reason}` в случае ошибки. `IoDevice` — это некий дескриптор, который используется для доступа к файлу.

```
@spec io:read(IoDevice, Prompt) => {ok, Term} | {error, Why} | eof
```

Читает терм Эрланга из `IoDevice`. Подсказка `Prompt` игнорируется если `IoDevice` представляет собой открытый файл. Подсказка `Prompt` используется для выдачи подсказки только если мы используем `io:read` для чтения стандартного ввода.

```
@spec file:close(IoDevice) => ok | {error, Why}
```

Закрывает `IoDevice`.

Используя эти функции мы можем реализовать `file:consult`, который мы использовали в предыдущей части. Вот, как `file:consult` может быть определён:

Загрузить [lib_misc.erl](#)

```
consult(File) ->
    case file:open(File, read) of
        {ok, S} ->
            Val = consult1(S),
            file:close(S),
            {ok, Val};
        {error, Why} ->
            {error, Why}
    end.

consult1(S) ->
    case io:read(S, '') of
        {ok, Term} -> [Term|consult1(S)];
        eof          -> [];
        Error        -> Error
    end.
```

На самом деле `file:consult` определён не так. Стандартная библиотека использует улучшенную обработку ошибок.

Ну а теперь пришло время посмотреть на версию из стандартной библиотеки. Если вы поняли, как работает предыдущая версия функции, то вы легко поймёте код из библиотеки. Вот только есть одна проблема. Как найти исходный код для `file.erl`?

Для нахождения кода мы используем функцию `code:which`, которая обнаруживает объектный код для любого загруженного модуля.

```
1> code:which(file).
"/usr/local/lib/erlang/lib/kernel-2.11.2/ebin/file.beam"
```

В стандартном релизе у каждой библиотеки есть две поддиректории. Одна, называемая `src`, содержит исходный код. Другая, называемая `ebin`, содержит скомпилированный Эрланг код. Так что исходный код для файла `file.erl` должен находиться в следующей директории:

```
/usr/local/lib/erlang/lib/kernel-2.11.2/src/file.erl
```

В случае, когда ничего уже не помогает, а документация не даёт ответов на ваши вопросы, быстрый взгляд в исходный код может помочь с ответом. Я знаю, что этого (поиска ответа в исходниках) не должно происходить, но все мы люди и иногда документация просто не помогает.

Чтение строк из файла по одной за раз

Если заменить `io:read` на `io:get_line`, то мы можем прочитать строки из файла по одной за раз. `io:get_line` читает символы до тех пор, пока не встретит символ перевода строки или конец файла. Вот пример:

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.43.0>}
2> io:get_line(S, '').
"{person, \"joe\", \"armstrong\",\\n"
3> io:get_line(S, '').
"\\t[\"occupation, programmer\",\\n"
4> io:get_line(S, '').
"\\t {favoriteLanguage, erlang}].\\n"
5> io:get_line(S, '').
"\\n"
6> io:get_line(S, '').
"{cat, {name, \"zorro\"}},\\n"
7> io:get_line(S, '').
" {owner, \"joe\"}].\\n"
8> io:get_line(S, '').
eof
9> file:close(S).
ok
```

Чтение всего файла целиком как бинарный объект

Вы можете использовать `file:read_file(File)`, чтобы прочитать файл целиком в бинарный объект, используя следующую атомарную операцию:

```
1> file:read_file("data1.dat").  
{ok,<<"{person, \\"joe\\", \\"armstrong\\\"...}>>}
```

`file:read_file(File)` возвращает `{ok, Bin}` в случае успеха и `{error, Why}` в противном случае.

Это явно лучший способ чтения файлов и я использую этот способ наиболее часто. В большинстве случаев я читаю файл целиком в память одной операцией, работаю над содержимым файла и сохраняю файл тоже одной операцией (используя `file:write_file`). У нас будет пример для данного способа работы.

Чтение файла с произвольным доступом

Если файл, который мы хотим прочитать, очень большой или содержит бинарные данные в формате, который определён где-то вовне, то мы можем открыть файл в сыром (raw) режиме и читать порции файла операцией `file:pread`.

Пример:

```
1> {ok, S} = file:open("data1.dat", [read,binary,raw]).  
{ok,{file_descriptor,prim_file,{\'#Port<0.106>,5}}}  
2> file:pread(S, 22, 46).  
{ok,<<"rong\\",\\n\\t[{occupation, progr...}>>}  
3> file:pread(S, 1, 10).  
{ok,<<"person, \"j">>}  
4> file:pread(S, 2, 10).  
{ok,<<"erson, \"jo">>}  
5> file:close(S).
```

`file:pread(IoDevice, Start, Len)` читает точно `Len` байт из `IoDevice`, начиная с байта в позиции `Start` (байты в файле нумеруются так, что первый байт находится в позиции 1) (прим. перев. - так в книге. В документации — всё по-другому). Она возвращает `{ok, Bin}` или `{error, Why}`.

В заключение, мы используем функции для произвольного доступа к файлу для написания утилиты, которая нам понадобится в следующей главе. В части 14.7 «Широковещательный сервер» мы разработаем простой широковещательный сервер (это сервер для так называемого потокового вещания. В данном случае — для вещания MP3). Часть этого сервера нуждается в поиске исполнителя и названий композиций, которые внедрены в файл MP3. Мы сделаем это в следующей части.

Чтение тегов MP3

MP3 — это бинарный формат, используемый для хранения сжатых звуковых данных. MP3 файлы сами по себе не содержат информацию о содержимом файла. К примеру в MP3 файле с музыкой не будет имени исполнителя. Эти данные (название композиции, исполнитель и прочее) хранятся внутри MP3 файла в специальном блочном формате ID3. Теги ID3 были придуманы программистом Eric Kemp для хранения метаданных, описывающих содержимое звукового файла. Вообще-то, есть несколько форматов ID3, но для наших целей мы будем использовать простейшие формы тегов — ID3v1 и ID3v1.1.

У тега ID3v1 простая структура — это последние 128 байт файла, содержащие тег фиксированной длины. Первые 3 байта содержат ASCII символы TAG, за которыми идут ряд полей фиксированной длины. Полностью эта 128 байтовая структура показана далее:

Длина	Содержимое
3	Заголовок, содержащий символы TAG
30	название
30	исполнитель
30	альбом
4	год
30	комментарий
1	жанр

В теге ID3v1 не было места, чтобы добавить номер композиции. Способ, реализующий это был предложен Michael Mutschler в формате ID3v1.1. Идея в том, чтобы заменить 30 байтовый комментарий следующим:

| Длина | Содержимое | | 28 | Комментарий | | 1 | 0 (ноль) | | 1 | Номер композиции |

Легко написать программу, которая будет читать теги ID3v1 из MP3 файла и сопоставлять поля, используя бинарное битовое сопоставление с образцом. Вот эта программа:

[Загрузить id3_v1.erl](#)

```
-module(id3_v1).
-import(lists, [filter/2, map/2, reverse/1]).
-export([test/0, dir/1, read_id3_tag/1]).

test() -> dir("/home/joe/music_keep" ).
```

```

dir(Dir) ->
    Files = lib_find:files(Dir, "*.mp3" , true),
    L1 = map(fun(I) ->
        {I, (catch read_id3_tag(I))}%
    end, Files),
%% L1 = [{File, Parse}] where Parse = error | [{Tag,Val}]
%% we now have to remove all the entries from L where
%% Parse = error. We can do this with a filter operation
L2 = filter(fun({_,error}) -> false;
            (_) -> true
        end, L1),
lib_misc:dump("mp3data" , L2).

read_id3_tag(File) ->
    case file:open(File, [read,binary,raw]) of
        {ok, S} ->
            Size = filelib:file_size(File),
            {ok, B2} = file:pread(S, Size-128, 128),
            Result = parse_v1_tag(B2),
            file:close(S),
            Result;
        Error ->
            {File, Error}
    end.

parse_v1_tag(<<$T,$A,$G,
             Title:30/binary, Artist:30/binary,
             Album:30/binary, _Year:4/binary,
             _Comment:28/binary, 0:8,Track:8,_Genre:8>>) ->

    {"ID3v1.1" ,
     [{track,Track}, {title,trim>Title)}, {artist,trimArtist)}, {album, trimAlbum)}]}];
parse_v1_tag(<<$T,$A,$G,
             Title:30/binary, Artist:30/binary,
             Album:30/binary, _Year:4/binary,
             _Comment:30/binary,_Genre:8>>) ->
    {"ID3v1" ,
     [{title,trim>Title)}, {artist,trimArtist)}, {album, trimAlbum)}]}];
parse_v1_tag(_) ->
    error.

trim(Bin) ->

```

```
list_to_binary(trim_blanks(binary_to_list(Bin))).  
  
trim_blanks(X) -> reverse(skip_blanks_and_zero(reverse(X))).  
  
skip_blanks_and_zero([$\\s|T]) -> skip_blanks_and_zero(T);  
skip_blanks_and_zero([0|T]) -> skip_blanks_and_zero(T);  
skip_blanks_and_zero(X) -> X.
```

Основная точка входа нашей программы — это `id3_v1:dir(Dir)`. Первое, что мы делаем — это ищем все наши MP3 файлы, вызывая `lib_find:find(Dir, "*.mp3", true)` (утилита поиска показана далее в части 13.8), которая рекурсивно сканирует директории ниже `Dir` на предмет файлов MP3. Найдя файл, мы разбираем теги, вызывая `read_id3_tag`. Разбор сильно упрощён, потому что мы используем простое битовое сопоставление с образцом. После этого мы подчищаем имена исполнителей и названия композиций, удаляя завершающие пробелы и нулевые символы, которые разделяют строки. В конце мы выводим результат в файл для дальнейшего использования (`lib_misc:dump` описывается в части E.2, Техника отладки).

Большинство музыкальных файлов помечены тегами ID3v1, даже если они дополнительно содержат ещё и теги стандартов ID3v2, v3, v4, добавленные позже, отформатированные по-другому и находящиеся в начале файла (или, что более редко — в середине файла). Программы для тегирования часто добавляют как ID3v1, так и дополнительные (и более трудные для чтения) теги в начало файла. Для наших целей мы сосредоточимся только на файлах, содержащих корректные теги ID3v1 и ID3v1.1.

Теперь, когда мы знаем, как читать файл, мы можем перейти к различным способам записи файла.

13.3 Разные способы записи файлов

Запись в файл включает в себя достаточно много таких же операций, как и чтение файла. Рассмотрим их подробнее.

Запись списка термов в файл

Предположим, что мы хотим создать файл, который мы сможем прочитать функцией `file:consult`. Стандартная библиотека вообще-то не содержит такой функции, так что мы напишем свою собственную. Назовём эту функцию `unconsult`.

Загрузить [lib_misc.erl](#)

```
unconsult(File, L) ->
```

```
{ok, S} = file:open(File, write),
lists:foreach(fun(X) -> io:format(S, "\~p.\~n", [X]) end, L),
file:close(S).
```

Мы можем выполнить это из оболочки Эрланга, чтобы создать файл, называемый `test1.dat`:

```
1> lib_misc:unconsult("test1.dat",
[{cats,["zorrow","daisy"]},
{weather,snowing}]).  
ok
```

Удостоверимся, что это действительно ОК:

```
2> file:consult("test1.dat").  
{ok,[{cats,["zorrow","daisy"]},{weather,snowing}]}
```

Чтобы реализовать `unconsult` мы открываем файл на запись и затем используем `io:format(S, "\~p.\~n", [X])` для записи термов в файл. `io:format` — это рабочая лошадка для создания форматированного вывода. Для выполнения форматированного вывода мы вызываем функцию:

```
@spec io:format(IoDevice, Format, Args) -> ok
```

`IoDevice` — это некое устройство ввода-вывода (которое было открыто в режиме записи), `Format` — это строка, содержащая коды форматирования, а `Args` — это список элементов для вывода.

Для каждого элемента из `Args` в строке формата должна присутствовать команда форматирования. Команды форматирования начинаются с тильды `\~`.

Вот некоторые наиболее часто используемые команды форматирования:

`\~n` - Перевод строки. `\~n` достаточно умён, так что работает платформонезависимо — на Unix — выведет в поток вывода ASCII (10), а на Windows — ASCII (13, 10)

`\~p` - Структурная распечатка аргумента

`\~s` - Аргумент является строкой

`\~W` - Вывод данных со стандартным синтаксисом. Используется для вывода термов Эрланга

У форматной строки есть масса аргументов, которые никто не будет запоминать в здравом уме. Как говорил Эйнштейн — для констант есть справочники. А для полного

списка параметров формата есть руководство по модулю `io`. Я помню только `~p`, `~s` и `~n`. Если вы начнёте с них, у вас не возникнет лишних проблем.

Лирическое отступление

Я соврал. Вам наверняка понадобится больше, чем просто `~p`, `~s`, `~n`. Вот пара примеров:

Формат	Результат
<code>io:format("~10s~n", ["abc"])</code>	<code>abc </code>
<code>io:format("~10s~n", ["abc"])</code>	<code> abc..... </code>
<code>io:format("~10.3.+s~n", ["abc"])</code>	<code> ++++++abc </code>
<code>io:format("~10.10.+s~n", ["abc"])</code>	<code> abc++++++ </code>
<code>io:format("~10.7.+s~n", ["abc"])</code>	<code> +++abc+++ </code>

Запись строк в файл

Это похоже на предыдущий пример — мы просто используем другие команды форматирования:

```
1> {ok, S} = file:open("test2.dat", write).
{ok,<0.62.0>}
2> io:format(S, "\~s\~n", ["Hello readers"]).
ok
3> io:format(S, "\~w\~n", [123]).
ok
4> io:format(S, "\~s\~n", ["that's it"]).
ok
5> file:close(S).
```

Это создаёт файл называемый `test2.dat` со следующим содержимым:

```
Hello readers
123
that's it
```

Запись всего файла целиком одной операцией

Это наиболее эффективный способ записи в файл. Функция `file:write_file(File, IO)` записывает данные IO (который является списком ввода-вывода, т. е. списком, элементами которого могут быть другие списки ввода-вывода, бинарные данные, целые числа от 0 до 255) в файл `File`. При записи список автоматически плоскится

(делается плоским — `flattened`, т. е. все квадратные скобки устраняются). Этот способ крайне эффективен и я этим частенько пользуюсь. Программа в следующей части демонстрирует это.

Вывод URL-ов из файла

Давайте напишем простенькую функцию, называемую `urls2htmlFile(L, File)`, которая берет список URL-ов `L` и создаёт HTML файл, где URL-ы представлены в виде кликабельных ссылок. Это позволит нам отработать технику создания целого файла одной-единственной операцией ввода-вывода.

Мы поместим нашу программу в модуль `scavenger_url`.

Загрузить `scavenger_urls.erl`

```
-module(scavenger_urls).
-export([urls2htmlFile/2, bin2urls/1]).
-import(lists, [reverse/1, reverse/2, map/2]).

urls2htmlFile(Urls, File) ->
    file:write_file(File, urls2html(Urls)).

bin2urls(Bin) -> gather_urls(binary_to_list(Bin), []).
```

В программе две точки входа. `urls2htmlFile(Urls, File)` берёт список URL-ов и создаёт HTML файл, содержащий кликабельные ссылки для каждого URL.

`bin2urls(Bin)` ищет по бинарным данным и возвращает список всех URL-ов, содержащихся в этих данных. Вот `urls2htmlFile`:

Загрузить `scavenger_urls.erl`

```
urls2html(Urls) -> [h1("Urls" ),make_list(Urls)].

h1>Title) -> ["<h1>" , Title, "</h1>\n"].

make_list(L) ->
    ["<ul>\n" ,
     map(fun(I) -> ["<li>" ,I,"</li>\n" ] end, L),
     "</ul>\n" ].
```

Этот код возвращает вложенный список символов. Заметьте, что мы не делали попыток сплющить список (что было бы довольно неэффективно). Мы создали вложенный список символов и просто отправили его в функцию вывода. Когда мы записываем вложенный список в файл функцией `file:write_file` система ввода-

вывода автоматически плющит список (т. е. записывает только символы из списка, но не скобки, создающие структуры списка). Ну и в конце — код, извлекающий URL-ы из бинарных данных:

Загрузить [scavenge_urls.erl](#)

```
gather_urls("<a href" ++ T, L) ->
    {Url, T1} = collect_url_body(T, reverse("<a href" )),
    gather_urls(T1, [Url|L]);
gather_urls([_], L) ->
    gather_urls(T, L);
gather_urls([], L) ->
    L.

collect_url_body("</a>" ++ T, L)      -> {reverse(L, "</a>"), T};
collect_url_body([H|T], L)            -> collect_url_body(T, [H|L]);
collect_url_body([], _)              -> {[[]], []}.
```

Чтобы выполнить это, нам надо иметь данные для разбора. Входные данные (бинарные данные) — это содержимое HTML страницы, так что нам нужна HTML страница для очистки от мусора. Для этого мы используем [socket_examples:nano_get_url](#) (см. главу 14.1, извлечение данных с сервера). Будем делать это по шагам в оболочке Эрланга:

```
1> B = socket_examples:nano_get_url("www.erlang.org"),
L = scavenge_urls:bin2urls(B),
scavenge_urls:urls2htmlFile(L, "gathered.html").
ok
```

Это создаст файл [gathered.html](#):

Загрузить [gathered.html](#)

```
<h1>Urls</h1>
<ul>
  <li><a href="old_news.html" >Older news.....</a></li>
  <li>
    <a href="http://www.erlang-consulting.com/training_fs.html">
      here</a>
  </li>
  <li><a href="project/megaco/" >Megaco home</a></li>
  <li><a href="EPLICENSE" >Erlang Public License (EPL)</a></li>
  <li><a href="user.html\#smtp_client-1.0">smtp_client-1.0</a></li>
  <li><a href="download-stats/" >download statistics graphs</a></li>
```

```
<li><a href="project/test_server" >Erlang/OTP Test Server</a></li>
<li><a href="http://www.erlang.se/euc/06/" >proceedings</a></li>
<li><a href="/doc/doc-5.5.2/doc/highlights.html" >
    Read more in the release highlights.
</a></li>
<li><a href="index.html" ></a></li>
</ul>
```

Запись файлов с произвольным доступом

Запись в файл с произвольным доступом подобна чтению. Сначала мы должны открыть файл в режиме записи. Затем мы используем `file:pwrite(Position, Bin)` для записи в файл. Вот пример:

```
1> {ok, S} = file:open("...", [raw, write, binary])
{ok, ...}
2> file:pwrite(S, 10, <<"new">>)
ok
3> file:close(S)
ok
```

Этот код записывает символы "new", начиная со смещения 10 в файле, перезаписывая имеющееся содержимое файла.

Операции над директориями

Для операций над директориями в модуле `file` есть три функции. `list_dir(Dir)` используется для получения списка файлов в `Dir`, `make_dir(Dir)` создаёт новую директорию и `del_dir(Dir)` удаляет директорию.

Если мы выполним `list_dir` в директории с кодом, которую я использую при написании этой книги, то мы увидим следующее:

```
1> cd("/home/joe/book/erlang/Book/code").
/home/joe/book/erlang/Book/code
ok
2> file:list_dir(".").
{ok,[{"id3_v1.erl"\~,",
"update_binary_file.beam",
"benchmark_assoc.beam",
"id3_v1.erl",
"scavenge_urls.beam",
"benchmark_mk_assoc.beam",
"benchmark_mk_assoc.erl",
```

```
"id3_v1.beam",
"assoc_bench.beam",
"lib_misc.beam",
"benchmark_assoc.erl",
"update_binary_file.erl",
"foo.dets",
"big.tmp",
..
```

Заметьте, что файлы в списке никак не упорядочены, никак не видно признаков, что данное имя является файлом или директорией, нет длин, вообще ничего нет.

Чтобы найти больше информации об индивидуальном файле в директории мы используем функцию `file:read_file_info`, которая подробнее описывается в следующей части.

Поиск информации о файле

Для нахождения информации о файле `F` мы вызываем функцию `file:read_file_info(F)`. Она возвращает `{ok, Info}`, если `F` — это правильное имя файла или директории. `Info` — это запись (record) типа `#file_info`, которая определена так:

```
-record(file_info,
        {size,                      % Размер файла в байтах
         type,                      % Атом: device, directory, regular, other
         access,                     % Атом: read, write, read_write, none
         atime,                      % Локальное время последнего чтения файла
         mtime,                      % Локальное время последней записи файла
         ctime,                      % Интерпретация этого поля зависит от
                                     % операционной
                                     % системы. В Unix это время последнего
                                     % изменения файла
                                     % или inode. В Windows – это время создания
                                     % файла
         mode,                      % Целое число: права на файл. В Windows права
                                     % владельца
                                     % будут дублироваться для группы и пользователя
         links,                      % Количество ссылок на файл (1, если файловая
                                     % система не поддерживает ссылки)
         major_device,               % Целое число: показывает файловую систему
                                     % (в Unix) или номер устройства
                                     % (A: = 0, B: = 1) (Windows)
```

Замечание: поля прав `mode` и доступа `access` перекрываются. Вы можете

использовать права, чтобы установить несколько файловых атрибутов одной операцией. Впрочем, вы можете использовать `access` для простых операций.

Чтобы найти длину и тип файла мы вызываем функцию `read_file_info` (заметьте, что нам приходится подключать `file.hrl`, который содержит определение записи `#file_info`):

Загрузить `lib_misc.erl`

```
-include_lib("kernel/include/file.hrl").  
file_size_and_type(File) ->  
    case file:read_file_info(File) of  
        {ok, Facts} ->  
            {Facts\#file_info.type, Facts\#file_info.size};  
        _ ->  
            error  
    end.
```

Теперь можно слегка улучшить вид списка, выведенного функцией `list_file`, добавив информацию о файлах в функции `ls()`:

Загрузить `lib_misc.erl`

```
ls(Dir) ->  
{ok, L} = file:list_dir(Dir),  
map(fun(I) -> {I, file_size_and_type(I)} end, sort(L)).
```

Теперь список отсортирован и в добавок содержит полезную информацию:

```
1> lib_misc:ls(".").  
[{"Makefile",{regular,1244}},  
 {"README",{regular,1583}},  
 {"abc.erl",{regular,105}},  
 {"alloc_test.erl",{regular,303}},  
 ...  
 {"socket_dist",{directory,4096}}]  
...
```

Дополнительное удобство в том, что модуль `filelib` экспортирует несколько маленьких функций, таких как `file_size(File)` и `is_dir(X)`. Это просто интерфейсы к `file:read_file_info`. Если нам надо всего лишь размер файла, то проще вызвать `filelib:file_size`, чем `file:read_file_info` и распаковывать элементы записи `#file_info`.

Копирование и удаление файлов

`file:copy(Source, Destination)` копирует файл `Source` в файл `Destination`.

`file:delete(File)` удаляет файл `File`.

Всякая всячина

К текущему моменту мы упомянули ряд функций, которые я ежедневно использую для манипулирования файлами. И крайне редко мне приходится обращаться к документации за дополнительной информацией. Что же я пропустил такого, что может вам понадобиться? Я приведу краткий обзор основных вещей. А за подробными деталями обращайтесь к документации.

Режим файла: когда мы открываем файл функцией `file:open`, мы открываем его в определённом режиме или комбинацией режимов. Вообще-то есть много разных режимов. К примеру, можно открыть файл на чтение и запись сжатого gzip файла. Ну и т. д. Полный список как обычно находится в документации.

Время модификации, группы, ссылки: мы можем установить всё это функциями из `file`.

Коды ошибок: я опрометчиво сказал, что у всех ошибок вид `{error, Why}`. На самом деле `Why` — это атом (к примеру, `enoent` означает, что файл не существует и т. д.) — есть большое количество кодов ошибок и все они описаны в документации.

`filename`: модуль `filename` содержит некоторые полезные функции для сбора полных имён файлов и директорий, поиска расширений файлов и прочего, а также для построения имён файлов из компонентов пути. Всё это делается платформонезависимым образом.

`filelib`: модуль `filelib` содержит небольшое количество функций, которые помогают сэкономить нам время. Например, `filelib:ensure_dir(Name)` обеспечивает, что все родительские директории для данного файла или директории существуют, создавая их при необходимости.

Программа поиска

И как финальный пример, мы используем `file:list_dir` и `file:read_file_info` для создания программы поиска общего назначения.

Главная точка входа в этот модуль следующая:

```
lib_find:files(Dir, RegExp, Recursive, Fun, Acc0)
```

Аргументы для неё:

`Dir` — имя директории, откуда начинать поиск файла

`RegExp` — регулярное выражение для проверки имени найденного файла. Если файлы, которые мы встретим, совпадают с этим регулярным выражением, то вызывается функция `Fun(File, Acc)`, где `File` — это имя файла, которое успешно сопоставлено с регулярным выражением.

`Recursive = true | false` — это признак, который определяет будет ли поиск заходить в поддиректории текущей директории.

`Fun(File, AccIn) -> AccOut` — это функция, которая применяется к файлу, если имя файла соответствует регулярному выражению `RegExp`. Начальное значение аккумулятора `Acc` — это `Acc0`. Каждый раз, когда вызывается `Fun`, она должна вернуть новое значение аккумулятора, которое будет передано в `Fun` при следующем вызове этого `Fun`. Конечное значение аккумулятора — это значение, возвращаемое из функции `lib_find:files/5`.

Мы можем передать в `lib_find:files/5` любую функцию, какую только захотим. Например, мы можем построить список файлов, используя следующую функцию, передавая ей в начале пустой список:

```
fun(File, Acc) -> [File|Acc] end
```

Точка входа модуля `lib_find:files(Dir, ShelRegExp, Flag)` обеспечивает упрощённый вызов для более общего использования программы. `ShelRegExp` здесь — это упрощённое регулярное выражение, которое легче записать, чем полную форму регулярного выражения.

Пример такой короткой формы записи:

```
lib_find:files(Dir, "*.erl" , true)
```

рекурсивно ищет все файлы Эрланга, начиная с `Dir`. Если бы последний аргумент был `false`, то программа искала бы файлы Эрланга только в директории `Dir`, но не спускалась в поддиректории.

Итак, код:

Загрузить [lib_find.erl](#)

```
-module(lib_find).  
-export([files/3, files/5]).
```

```

-import(lists, [reverse/1]).  

-include_lib("kernel/include/file.hrl" ).  

files(Dir, Re, Flag) ->  

    Re1 = regexp:sh_to_awk(Re),  

    reverse(files(Dir, Re1, Flag, fun(File, Acc) ->[File|Acc] end, [])).  

files(Dir, Reg, Recursive, Fun, Acc) ->  

    case file:list_dir(Dir) of  

        {ok, Files} -> find_files(Files, Dir, Reg, Recursive, Fun, Acc);  

        {error, _} -> Acc  

    end.  

find_files([File|T], Dir, Reg, Recursive, Fun, Acc0) ->  

    FullName = filename:join([Dir,File]),  

    case file_type(FullName) of  

        regular ->  

            case regexp:match(FullName, Reg) of  

                {match, _, _} ->  

                    Acc = Fun(FullName, Acc0),  

                    find_files(T, Dir, Reg, Recursive, Fun, Acc);  

                _ ->  

                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)  

            end;  

        directory ->  

            case Recursive of  

                true ->  

                    Acc1 = files(FullName, Reg, Recursive, Fun, Acc0),  

                    find_files(T, Dir, Reg, Recursive, Fun, Acc1);  

                false ->  

                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)  

            end;  

        error ->  

            find_files(T, Dir, Reg, Recursive, Fun, Acc0)  

    end;  

find_files([], _, _, _, _, A) ->  

    A.  

file_type(File) ->  

    case file:read_file_info(File) of  

        {ok, Facts} ->  

            case Facts\#file_info.type of  

                regular -> regular;  

                directory -> directory;

```

```
    _ -> error
end;
_ ->
error
end.
```

Глава 14. Программирование с сокетами

Наиболее интересные программы, которые я пишу так или иначе включают сокеты.

Сокет – это конечная точка соединения, которая позволяет взаимодействовать машинам по Интернет, используя Internet Protocol(IP). В этом разделе мы сконцентрируем свое внимание на двух протоколах интернета: Transmission Control Protocol(TCP) и User Datagram Protocol(UDP)

UDP позволяет приложениям посыпать друг другу короткие сообщения(называемые дейтаграммами), но этот протокол не гарантирует доставку сообщений. Дейтаграммы могут прийти в неправильном порядке. С другой стороны – TCP, предоставляет надежный поток байтов, которые доставляются в правильном порядке на протяжении всего соединения.

Почему программирование с использованием сокетов увлекательно? Потому что это позволяет приложениям взаимодействовать по интернет с другими машинами, что имеет гораздо больший потенциал, чем выполнение локальных операций.

Существуют две основные библиотеки для программирования на сокетах – это `get_tcp`, для программирования TCP соединений, и `gen_udp` для UDP соединений

В этой главе мы увидим как клиент и сервер используют TCP и UDP сокеты. Мы пройдем через различные формы серверов: параллельные, последовательные, блокирующий и неблокирующие, и увидим, как сделать traffic-shaping приложения, которые контролируют поток данных к программам.

14.1 Использование TCP

Мы начнем наше путешествие в программирование сокетов с рассмотрения простой TCP программы, которая получает данные с сервера. После этого мы напишем простой последовательный TCP сервер, и покажем, как он может быть распараллелен для обработки множества параллельных сессий.

14.1.1 Получение данных с сервера

Начнем с написания небольшой функции (1)(2), которая использует TCP сокет для получения HTML страницы с <http://www.google.com>:

Загрузить [socket_examples.erl](#)

```
nano_get_url() ->
    nano_get_url("www.google.com").

nano_get_url(Host) ->
    {ok,Socket} = gen_tcp:connect(Host,80,[binary, {packet, 0}]),
    ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
    receive_data(Socket, []).

receive_data(Socket, SoFar) ->
    receive {tcp,Socket,Bin} ->
        receive_data(Socket, [Bin|SoFar]);
{tcp_closed,Socket} -> list_to_binary(reverse(SoFar))
end.
```

(1) стандартная библиотечная функция, которая делает то же самое называется `http:request(Url)`. Но мы хотим показать, как это можно сделать средствами библиотеки `gen_tcp`.

(2) В современной версии документации нету библиотеки `http`, зато есть `httpc`

Как это работает?

1. Мы открываем TCP сокет с адресом `http://google.com` на 80 порту, при помощи `gen_tcp:connect`. Аргумент `binary` в функции `connect` говорит системе открыть сокет в двоичном режиме и доставлять все данные приложению как бинарные. `{packet,0}` в контексте TCP означает, что данные доставляются непосредственно приложению, в немодифицированной форме.
2. Мы вызываем `get_tcp:send` и посылаем сообщение `GET / HTTP/1.0\r\n\r\n` в сокет. Затем мы ожидаем ответа. Ответ не придет весь одним пакетом, он прийдет фрагментами. Процесс, открывший сокет, или контролирующий его будет получать фрагменты, как последовательность сообщений.
3. Мы принимаем сообщение вида `{tcp,Socket,Bin}`. Третий аргумент этого кортежа – это двоичные данные. Так получилось потому, что мы открыли сокет в бинарном режиме. Это сообщение – один из фрагментов, которые веб-сервер посыпает нам. Мы получили один фрагмент, добавили его в список фрагментов, и затем ожидаем следующий фрагмент.
4. Мы получаем `{tcp_closed,Socket}`. Это произошло потому, что сервер закончил отправку данных. (3)

5. Когда все фрагменты пришли, нам необходимо развернуть список, поскольку мы сохраняли фрагменты в неправильном порядке.

Давайте проверим, что это работает:

```
1> B = socket_examples:nano_get_url().  
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\nCache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"...>>
```

Примечание: Когда вы запускаете `nano_get_url`, то результат будет двоичный. Таким образом вы увидите, что двоичные данные выглядят, как при "pretty printed" в эрланговской оболочке. Когда двоичные данные печатаются в формате "pretty printed" все управляющие символы выводятся в escape-формате. Бинарные данные выводятся не полностью, это видно по трем точкам (`...>>`) в конце печати. Если вы желаете увидеть все бинарные данные, можно использовать `io:format`, или разорвать бинарные данные на символы, при помощи `string:tokens`:

```
2> io:format("~p~n",[B]).  
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\nCache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"  
     TM=176575171639526:LM=1175441639526:S=gkfTrK6AFkybT3;  
     expires=Sun, 17-Jan-2038 19:14:07  
     ... several lines omitted ...  
>>  
  
3>string:tokens(binary_to_list(B),"\r\n").  
[ "HTTP/1.0 302 Found",  
  "Location: http://www.google.se/",  
  "Cache-Control: private",  
  "Set-Cookie: PREF=ID=ec7f0c7234b852dece4:TM=11713424639526:  
     LM=1171234639526:S=gsdertTrK6AEybT3;  
  expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com",  
  "Content-Type: text/html",  
  "Server: GWS/2.1",  
  "Content-Length: 218",  
  "Date: Fri, 16 Feb 2007 15:25:26 GMT",  
  "Connection: Keep-Alive",  
  ... lines omitted ...
```

(3) Это верно только для HTTP/1.0; для более новых версий используются другие стратегии

Это более или менее показывает, как работает web-клиент (с уклонением в меньшую сторону — приходится делать много работы, для корректного вывода результата в

веббраузерах). Предыдущий код, тем не менее, — хорошая отправная точка для ваших собственных экспериментов. Вам может понравиться модифицировать этот код, например: вы можете принимать и хранить записи веб-сайтов или автоматически прочитать ваш ваш e-mail. Возможности безграничны.

Заметим, что код, который собирает фрагменты выглядел так:

```
receive_data(Socket, SoFar) ->
    receive {tcp,Socket,Bin} ->
        receive_data(Socket, [Bin|SoFar]);
    {tcp_closed,Socket} ->
        list_to_binary(reverse(SoFar)) end.
```

Таким образом мы добавляем прибывшие фрагменты в голову списка `SoFar`. Когда все фрагменты прибыли и сокет был закрыт мы реверсируем список и соединяем фрагменты.

Вы могли подумать, что сборка фрагментов таким образом было бы лучшим решением:

```
receive_data(Socket, SoFar) ->
    receive {tcp,Socket,Bin} ->
        receive_data(Socket, list_to_binary([SoFar,Bin]));
    {tcp_closed,Socket} ->
        SoFar end.
```

Этот код корректен, но менее эффективен, чем оригинальная версия. Причина этого состоит в том, что этот код беспрестанно добавляет новые данные в конец буфера, а это включает большое копирование данных. Куда лучше добавлять фрагменты в голову списка, а потом реверсировать записи списка и собрать все фрагменты одной операцией.

14.1.2 Простой TCP сервер

В предыдущем разделе мы написали простой клиент. Давайте теперь напишем сервер.

Сервер открывает порт 2345 и ждет одного сообщения. Это двоичное сообщение, которое содержит терм эрланга. Терм – это эрданговская строка, содержащая выражение. Сервер обрабатывает выражение и посыпает результат клиенту, записывая результат в сокет.

Как мы можем писать веб сервер?

Написание что-то вроде веб-клиента или сервера очень интересно. Действительно, многие люди уже имеют написанные эти вещи, но если вы действительно хотите

понять, как это работает, копайте глубже и выясните. Это очень поучительно. Кто знает — может быть наш веб-сервер будет самый лучший. Так как нам начать?

Для создания веб-сервера, или другого программного обеспечения, которое реализует один из стандартных протокол интернета, вам необходимы правильные инструменты, а так же необходимо четко понимать, как точно этот протокол реализовать.

В нашем примере есть код, который получает веб-страницу. Откуда мы узнали, что надо открывать 80-ый порт. Откуда мы узнали, что серверу надо посыпать именно такое сообщение: `GET / HTTP/1.0\r\n\r?\r`. Ответ прост. Все основные протоколы для интернет сервисов описаны в **request for comments(RFCs)**. **HTTP/1.0** описан в **RFC 1945**. Официальный веб-сайт для всех документов RFC — <http://www.ietf.org> (сайт Internet Engineering Task Force).

Другой бесценный источник информации - снiffeр. При помощи снiffeра мы можем захватывать и анализировать все IP пакеты приходящие и уходящие от приложения. Большое количество снiffeров включают программное обеспечение, которое может декодировать и анализировать данные в пакете, а так же представлять данные в выразительной форме. Один из наиболее известных и возможно лучших снiffeров — это Wireshark (Ранее известный, как ethereal), доступен на <http://www.wireshark.org>.

Вооружившись пакетным снiffeром и соответствующими документами RFC, мы готовы писать следующие убийственные приложения.

Написав эту программу, мы сможем ответить на несколько простых вопросов:

- Как организованы эти данные? Как мы узнаем, сколько данных составляет один запрос или ответ?
- Как эти данные кодируются и декодируются в пределах запроса или ответа?

Данные TCP сокета — это просто недифференцируемый поток байтов. В процессе доставки эти данные могут быть фрагментированы. Поэтому нам нужно некоторое соглашение, что бы мы знали сколько байтов составляет единичный запрос или ответ.

В случае Эрланга мы используем простое соглашение, по которому перед каждым запросом или ответом мы приписываем перед ним 1,2 или 4 байта, которые характеризуют его длину. Это количество байт передается функциям `get_tcp:connect` и `gen_tcp:listen`, как аргумент `{packet,N}` (4). Заметим, что аргумент `packet` должен быть согласован между клиентом и сервером. Если сервер откроет соединение с `{packet,2}`, а клиент с `{packet,4}`, то ничего работать не будет.

Имея открытый сокет с опцией `{packet, N}`, мы не должны беспокоиться о фрагментации данных. Драйвер эрланга ,до передачи сообщения нашей программе, удостоверится, что все фрагменты данных собраны с правильной длиной.

Следующий интерес представляют кодирование и декодирование данных. Мы будем использовать простейший возможный путь кодирования и декодирования сообщений, используя `term_to_binary` для кодирования и `binary_to_term` для декодирования.

Заметим, что соглашение упаковки `({packet, N})` и правила кодирования, необходимые для общения клиента и сервера, достигаются в двух строках кода: используя `{packet, 4}`, когда мы открываем сокет, и `term_to_binary` — для кодирования.

Легкость, с которой мы можем упаковывать и кодировать эрланговские термы, дает нам значительное преимущество над text-based методами, такими как HTTP или XML. Используя эрланговский BIF `term_to_binary` и его обратный `binary_to_term`, обычно, на порядок быстрее, чем вычисление эквивалентных операций, которые использует XML термы и включают пересылку намного большего количества данных. А теперь к программам. Во-первых, вот очень простой сервер.

Загрузить `socket_examples.erl`

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                         {reuseaddr, true},
                                         {active, true}]),
    {ok, Socket} = gen_tcp:accept(Listen),
    gen_tcp:close(Listen),
    loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Bin} ->
            io:format("Server received binary = ~p~n",[Bin]),
            Str = binary_to_term(Bin), %% (9)
            io:format("Server (unpacked) ~p~n",[Str]),
            Reply = lib_misc:string2value(Str), %% (10)
            io:format("Server replying = ~p~n",[Reply]),
            gen_tcp:send(Socket, term_to_binary(Reply)),
            loop(Socket);
        {tcp_closed, Socket} ->
            io:format("Server socket closed~n")
    end.
```

(4) Директива `packet` здесь означает не сколько физически байтов будет записываться в сокет, а именно длину сообщения в программе

Как это работает?

1. Вначале мы вызываем `gen_tcp:listen`, для прослушивания 2345 порта, и устанавливаем соглашение об упаковке. `{packet,4}` подразумевает, что сообщению будет предшествовать 4х байтовый заголовок. Затем `gen_tcp:listen(...)` возвращает `{ok, Socket}` или `{error, Why}`, но нас интересует только тот случай, когда мы можем открыть сокет. Поэтому мы пишем следующий код:

```
{ok, Listen} = gen_tcp:listen(...)
```

Здесь происходит сопоставление по шаблону, и если `get_tcp:listen` возвратит `{error, ...}`, то будет поднято исключение. В случае успеха, это выражение связывает `Listen` с прослушиваемым сокетом, и он используется в качестве аргумента в `gen_tcp:accept`

2. Теперь мы вызываем `gen_tcp:accept(Listen)`. В этом месте программа усыпляется и ожидает соединения. Когда соединение установлено, эта функция возвращает переменную `Socket`, связанную с сокетом, который может использоваться для общения с клиентом, который установил соединение.
3. Когда функция `accept` возвращает управление, мы сразу же вызываем `gen_tcp:close(Listen)`. Функция `close` закрывает прослушиваемый сокет, после чего сервер становится недоступным для других соединений. Это не оказывает эффекта на имеющееся соединение; это только предотвращает новые соединения.
4. Мы декодируем входные данные.
5. Затем мы вычисляем строку.
6. И наконец, мы кодируем ответ и посылаем его обратно в сокет.

Заметим, что эта программа принимает только один единственный запрос, потом она завершится и больше не будет принимать соединений.

Это простейший пример сервера, иллюстрирующий то, как упаковываются и коидрутся данные.

Этот код принимает запрос, вычисляет ответ, отправляет ответ, и завершается.

Для тестирования сервера нам потребуется соответствующий клиент:

[Загрузить socket_examples.erl](#)

```
nano_client_eval(Str) ->
{ok, Socket} =
    gen_tcp:connect("localhost", 2345,
                    [binary, {packet, 4}]),
    ok = gen_tcp:send(Socket, term_to_binary(Str)),
receive
{tcp,Socket,Bin} ->
    io:format("Client received binary = ~p~n",[Bin]),
    Val = binary_to_term(Bin),
    io:format("Client result = ~p~n",[Val]),
    gen_tcp:close(Socket)
end.
```

Для тестирования нашего кода мы запустим клиент и сервер на одной машине, поэтому адрес хоста жестко прописан в функции `gen_tcp:connect`, как `localhost`. Заметим, что `term_to_binary` вызывается клиентом для кодирования сообщения и `binary_to_term` вызывается сервером, для переконструирования пришедшего сообщения. Для запуска этого кода нам потребуется открыть два терминала и запустить эрланговскую оболочку в каждом из них. В начале мы запустим сервер:

```
1> socket_examples:start_nano_server().
```

Мы не увидим другого вывода, пока ничего не происходит. Затем перейдем к клиенту и напишем следующую команду:

```
1> socket_examples:nano_client_eval("list_to_tuple([2+3*4,10+20])"
```

В окне с сервером мы должны увидеть следующее:

```
Server received binary = <<131,107,0,28,108,105,115,116,95,116,
111,95,116,117,112,108,101,40,91,50,
43,51,42,52,44,49,48,43,50,48,93,41>>
Server (unpacked) "list_to_tuple([2+3*4,10+20])"
Server replying = {14,30}
```

В окне с клиентом мы должны увидеть такой текст:

```
Client received binary = <<131,104,2,97,14,97,30>>
Client result = {14,30}
ok
```

И в конце концов в серверном окне будет вот так:

```
Server socket closed
```

14.1.3 Улучшение сервера

В предыдущем разделе мы сделали сервер, который принимает только одно соединение, после чего завершается. Чуть-чуть изменив код мы можем получить два различных вида серверов:

1. Последовательный — принимает одно соединение единовременно.
2. Параллельный сервер — множество параллельных соединений единовременно.

Изначальная версия кода выглядит так:

```
start.nano_server() ->
  {ok, Listen} = gen_tcp:listen(...),
  {ok, Socket} = gen_tcp:accept(Listen),
  loop(Socket). ...
```

Будем изменять этот код, и получим два варианта серверов.

14.1.4 Последовательный сервер

Для создания последовательного сервера мы изменим код следующим образом:

```
start.seq_server() ->
  {ok, Listen} = gen_tcp:listen(...),
  seq_loop(Listen). seq_loop(Listen) ->
  {ok, Socket} = gen_tcp:accept(Listen),
  loop(Socket),
  seq_loop(Listen).

loop(..) -> %% по-старому
```

Этот код работает почти как и предыдущий, но так как мы хотим обрабатывать больше одного запроса, мы оставляем прослушиваемый сокет открытым и не вызываем `gen_tcp:close(Listen)`.

Другое отличие, что после того, как `loop(Socket)` завершится, мы вызываем `seq_loop(Listen)` снова, где и ожидается следующее соединение.

Если клиент попытается соединиться с сервером, пока он занят с имеющимся соединением, то тогда он будет поставлен в очередь, пока сервер не закончит обработку текущего соединения. Если число соединений в очереди будет превышать

значение `listen backlog`, тогда соединения будут отвергаться.

Мы показали код, который только запускает сервер. Останов сервера прост(как и останов параллельного); просто убейте процесс, который запускал сервер или серверы. `gen_tcp` связывает себя с контролируемыми процессами. И если контролируемый процесс умирает, то это закрывает сокет.

14.1.5 Параллельный сервер

Трюк создания параллельного сервера немедленно порождает дочерний процесс, когда `gen_tcp:accept` получает новое соединение:

```
start_parallel_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen) end),
    loop(Socket).

loop(..) -> %% как и раньше
```

Этот код схож с последовательным сервером, который вы видели ранее. Решающее различие заключается в добавлении функции `spawn`, которая гарантирует создание параллельных процессов, для каждого нового соединения. Вы должны взглянуть на место, где стоит `spawn` и увидеть, как эта функция превращает последовательный сервер в параллельный.

Все эти сервера вызывают `gen_tcp:listen` и `gen_tcp:accept`; единственное различие заключается в том, что мы называем эти функции параллельной программой или последовательной программой.

14.1.6 Заметки

Будем осведомлены о следующем:

- Процесс, который создает сокет (вызывая `gen_tcp:accept` или `gen_tcp:connect`) называется процессом, контролирующим этот сокет. Все сообщения из сокета будут отправляться контролирующему процессу; Если контролирующий процесс умирает, тогда сокет будет закрыт. Контролирующий процесс может быть изменен на `NewPid` при помощи вызова `gen_tcp:controlling_process(Socket, NewPid)`.
- Наш сервер потенциально может установить многие тысячи соединений. Возможно, мы захотим ограничить максимальное число одновременных

соединений. Это может быть реализовано при помощи счетчика того, сколько соединений сейчас установлено. Мы инкрементируем его, если поступает новое соединение, и декрементируем, если соединение завершается. Мы можем использовать этот механизм для ограничения общего числа одновременных соединений в системе.

- После принятия соединения хорошей идеей будет явное задание необходимых опций сокета, вот так:

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet,4},binary,
                      {nodelay,true},{active, true}]),
loop(Socket)
```

- В версии эрланга R11B-3 различным процессам позволено вызывать `gen_tcp:accept` для одного и того же сокета. Это простейший пример параллельного сервера, поскольку мы можем иметь кучу заранее порожденных процессов, каждый из которых будет ожидать соединения при помощи `gen_tcp:accept/1`.

14.2 Контроллирование проблемы

Эрланг-сокет может быть открыт в одном из трех режимов: *активный*, *единожды активный*, *пассивный*. Это достигается включением опции `{active,true|false|once}` в аргумент Options одной из двух функций `gen_tcp:connect(Address,Port,Options)` или `gen_tcp:listen(Port,Options)`.

Если указано `{active,true}`, тогда будет создан активный сокет; `{active,false}` указывает на создание пассивного сокета. `{active,once}` создает сокет, который будет активным, но только до приема одного сообщения; после того, как сообщение будет принято, сокет сделается пассивным до того момента, как сможет принять новое сообщение.

В следующих разделах мы посмотрим, как применяются эти различные виды сокетов. Различие между активным и пассивным режимом заключается в том, как происходит прием сообщения сокетом.

- Если был создан активный сокет, тогда контроллирующему процессу будут приходить кортежи вида `{tcp,Socket,Data}` в почтовый ящик. В этом случае контроллирующий процесс никак не сможет контролировать поток сообщений. Злоумышленник может отправить тысячи сообщений системе, и все они будут доставлены контроллирующему процессу. Контроллирующий процесс никак не сможет сортировать этот поток сообщений.

- Если сокет был открыт в пассивном режиме, тогда для приема сообщений с сокета контроллирующий процесс вызывает `gen_tcp:recv(Socket, N)`. Этот вызов будет пытаться получить ровно `N` байт из сокета. Если `N = 0`, тогда все доступные байты будут возвращены. В этом случае сервер может контролировать поток байтов от клиента, выбирая, когда использовать `gen_tcp:recv`.

Пассивный режим используется для контроля потока, который идет на сервер. Для иллюстрации этого мы можем написать функцию по приему сообщений в трех видах:

- Активный прием сообщений(неблокирующий)
- Пассивный прием сообщений(блокирующий)
- Гибридный прием сообщений(частичное блокирование)

14.2.1 Активный прием сообщений(неблокирующий)

В нашем первом примере мы открываем сокет в активном режиме и затем принимаем сообщения с сокета:

```
{ok, Listen} = gen_tcp:listen(Port, [...,{active, true}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... делаем что-то с Data ...
        {tcp_closed, Socket} ->
            ...
    end.
```

Этот процесс не может контролировать поток сообщений к серверу. Если клиент производит данные быстрее, чем сервер может обработать, тогда система может быть зафлужена(flooded) сообщениями — буфер сообщений заполнится и система может упасть или вести себя странно.

Этот тип сервера называется неблокирующим сервером, потому что он не может блокировать клиента. Мы должны писать неблокирующие сервера, только в том случае, когда мы можем быть уверены, что сервер сможет справиться с запросами клиентов.

14.2.2 Пассивный прием сообщений(блокирующий)

В этом разделе мы напишем блокирующий сервер. Сервер открывает сокет в

пассивном режиме, устанавливая опцию `{active, false}`. Сервер не может обрушиться из-за гиперактивного клиента, который пытается зафлудить его большим количеством данных.

Код в функции `loop` вызывает `gen_tcp:recv` все время, когда нужно принять данные. Клиент будет заблокирован, пока сервер вызывает `recv`. Заметим, что ОС тоже буферизует данные, что позволяет клиенту отправить большое количество данных пока он заблокирован, даже если `recv` не вызывалась.

```
{ok, Listen} = gen_tcp:listen(Port, [...,{active, false}...]),  
{ok, Socket} = gen_tcp:accept(Listen),  
loop(Socket).  
  
loop(Socket) ->  
    case gen_tcp:recv(Socket, N) of  
        {ok, B} ->  
            ... делаем что-то с данными ...  
            loop(Socket);  
        {error, closed}  
            ... end.
```

14.2.3 Гибридный подход(частичное блокирование)

Вы можете подумать, что использование пассивного режима для всех серверов — это корректный подход. К сожалению, это не так, когда мы были в пассивном режиме, мы могли ожидать данные только с одного сокета. Это не годится для написания серверов, которые должны ожидать данные со многих сокетов.

К счастью, мы можем применить гибридный подход, когда никто ни неблокирующий, ни блокирующий. Мы открываем сокет с опцией `{active, once}`. В этом режиме сокет активен, но только до первого сообщения. После того, как контролирующему процессу было послано сообщение, необходимо вызвать `inet:setops`, чтобы включить прием следующего сообщения. Система будет блокировать прием, пока это не произойдет. Это лучше, чем два других метода. Вот как выглядит этот код:

```
{ok, Listen} = gen_tcp:listen(Port, [...,{active, once}...]),  
{ok, Socket} = gen_tcp:accept(Listen),  
loop(Socket).  
  
loop(Socket) ->  
    receive  
        {tcp, Socket, Data} ->  
            ... do something with the data ...  
            %% когда вы готовы принять следующее сообщение
```

```
    inet:setopts(Sock, [{active, once}]), loop(Socket);
{tcp_closed, Socket} ->
    ...
end.
```

Используя `{active,once}` опцию пользователь может осуществлять продвинутые формы контроля потока (иногда это называется traffic-shaping) и ,таким образом, предотвращать зафлужевание(flooded) сервера чрезмерными сообщениями.

14.3 Откуда это соединение к нам пришло?

Допустим, мы написали некоторый вид онлайн сервера и заметили, что кто-то спамит наш сайт. Как мы можем на это отреагировать? Первым делом необходимо узнать, откуда поступило это соединение. Чтобы это определить мы можем использовать `inet:peername(Socket)`.

```
@spec inet:peername(Socket) -> {ok, {IP_Address, Port}} | {error, Why}
```

Эта функция возвращает IP адрес и порт другого конца соединения, таким образом сервер может определить кто инициировал соединение. IP_Adress это кортеж из целых чисел, для IPv4 имеет вид `{N1,N2,N3,N4}`, а для IPv6 `{K1,K2,K3,K4,K5,K6}`. Ni и Ki числа в диапазоне от 0 до 255.

14.4 Обработка ошибок сокетов

Обработка ошибок сокета это очень просто, по существу вам ничего не надо делать. Как говорилось ранее, каждый сокет имеет контроллирующий процесс(тоесть процесс, который создал сокет). Если контроллирующий процесс умирает, тогда сокет автоматически хакрывается.

Это значит, что если, например, вы имеете клиент и сервер, и сервер падает из-за программной ошибки, то сокет, который принадлежил серверу будет автоматически закрыт, и клиенту будет послан кортеж `{tcp_closed,Socket}`.

Мы можем протестировать этот механизм со следующий программой:

Загрузить [socket_examples.erl](#)

```
error_test() ->
    spawn(fun() -> error_test_server() end),
    lib_misc:sleep(2000),
```

```

{ok,Socket} = gen_tcp:connect("localhost",4321,[binary, {packet, 2}]),
io:format("connected to:~p~n",[Socket]),
gen_tcp:send(Socket, <<"123">>),
receive
    Any ->
        io:format("Any=~p~n",[Any]) end.

error_test_server() ->
{ok, Listen} = gen_tcp:listen(4321, [binary,{packet,2}]),
{ok, Socket} = gen_tcp:accept(Listen),
error_test_server_loop(Socket).

error_test_server_loop(Socket) ->
receive
    {tcp, Socket, Data} ->
        io:format("received:~p~n",[Data]),
        atom_to_list(Data),
        error_test_server_loop(Socket)
end.

```

Когда мы запустим ее, мы увидим следующее:

```

1> socket_examples:error_test().
connected to:#Port<0.152>
received:<<"123">>
=ERROR REPORT==== 9-Feb-2007::15:18:15 ===
Error in process <0.77.0> with exit value:
  {badarg,[{erlang,atom_to_list,[<<3 bytes>>]}, {socket_examples,error_test_server_loop,1}]}
Any={tcp_closed,#Port<0.152>}
ok

```

при помощи spawn мы породили сервер, затем усыпились на 2 секунды (чтобы сервер успел запуститься), и затем отправляем сообщение, содержащее <<"123">>. Когда это сообщение приходит, сервер пытается вычислить atom_to_list(Data), где Data — это бинарные данные, и немедленно падает(5). Теперь, когда контролирующий процесс(со стороны сервера) обрушился, сокет автоматически закрывается. Затем клиенту отправляется сообщение {tcp_closed,Socket}.

(5)системный монитор печатает диагностическое сообщение, которое вы видите в оболочке

14.5 UDP

Теперь давайте рассмотрим User Datagram Protocol (UDP). Используя UDP, машины могут отправлять друг другу коротенькие сообщения по интернет, которые называются дейтаграммы. UDP дейтаграммы ненадежны. Это значит что если клиент отправил последовательность UDP дейтаграмм серверу, то они могут прийти не в том порядке, в каком отправлялись, могут прийти не все, или дейтаграммы могут продублироваться, но если одна дейтаграмма пришла на сервер, то она будет неповрежденной. Большие дейтаграммы могут быть разбиты на маленькие фрагменты, но IP протокол будет собирать фрагменты, перед передачей приложению.

UDP устроен так, что до отправки дейтаграммы между клиентом и сервером не установлено соединение. Это значит, что UDP хорошо приспособлен для приложений, у которых большое число клиентов, которые отправляют короткие сообщения серверу.

Создание UDP клиента и сервера в Эрланге еще проще, чем создание клиента или сервера на TCP, если мы не беспокоимся о поддержке соединения.

14.5.1 Простейший UDP сервер и клиент

Давайте вначале обсудим сервер. Основной вид UDP сервера следующий:

```
server(Port) ->
{ok, Socket} = gen_udp:open(Port, [binary]),
loop(Socket).

loop(Socket) ->
receive
    {udp, Socket, Host, Port, Bin} ->
        BinReply = ... ,
        gen_udp:send(Socket, Host, Port, BinReply),
        loop(Socket)
end.
```

Этот код, отчасти, проще кода с TCP, поскольку мы не беспокоимся о приеме сообщения "socket closed". Заметим, что открыли сокет в бинарном режиме, что говорит драйверу отправлять все сообщения контролирующему процессу, как бинарные данные.

Теперь клиент. Тут просто открывается UDP сокет, отправляется сообщение серверу, ожидается ответ(или таймаут), и затем закрывается сокет и возвращается значение, которое пришло от сервера.

```
client(Request) ->
{ok, Socket} = gen_udp:open(0, [binary]),
ok = gen_udp:send(Socket, "localhost" , 4000, Request),
```

```

Value = receive
    {udp, Socket, _, _, Bin} ->
        {ok, Bin}
    after 2000 ->
        error
    end,
gen_udp:close(Socket),
Value

```

Мы должны выставить таймаут, поскольку UDP ненадежный, мы можем просто не получить ответа.

14.5.2 UDP факториал сервер

Мы легко можем создать модуль UDP сервера, который подсчитывает факториал любого целого числа, которого ему отправили. Код сделан по аналогии с предыдущим разделом.

[Загрузить udp_test.erl](#)

```

-module(udp_test).
-export([start_server/0, client/1]).

start_server() ->
    spawn(fun() -> server(4000) end).

%% Сервер
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    io:format("server opened socket:~p~n",[Socket]),
    loop(Socket).

loop(Socket) ->
    receive
        {udp, Socket, Host, Port, Bin} = Msg ->
            io:format("server received:~p~n",[Msg]),
            N = binary_to_term(Bin),
            Fac = fac(N),
            gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
            loop(Socket)
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).

```

```

%% Клиент
client(N) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    io:format("client opened socket=~p~n",[Socket]),
    ok = gen_udp:send(Socket, "localhost", 4000,
                        term_to_binary(N)),
    Value = receive
        {udp, Socket, _, _, Bin} = Msg ->
            io:format("client received:~p~n",[Msg]),
            binary_to_term(Bin)
    after 2000 ->
        0
    end,
    gen_udp:close(Socket),
    Value.

```

Обратите внимание, что я добавил несколько отладочных печатей, что бы вы увидели что происходит, когда запускается программа. Я всегда добавляю несколько отладочных печатей, когда разрабатываю программу, и затем комментирую или редактирую их во время работы программы.

Теперь давайте запустим этот пример. Вначале запустим сервер.

```

1> udp_test:start_server().
server opened socket:#Port<0.106>
<0.34.0>

```

Он запускается в фоновом режиме, теперь мы можем сделать клиентский запрос:

```

2> udp_test:client(40).
client opened socket=#Port<0.105>
server received:{udp,#Port<0.106>,{127,0,0,1},32785,<<131,97,40>>}
client received:{udp,#Port<0.105>,
                {127,0,0,1}, 4000,
                <<131,110,20,0,0,0,0,64,37,5,255,
                100,222,15,8,126,242,199,132,27,
                232,234,142>>}
815915283247897734345611269596115894272000000000

```

14.5.3 Дополнительные замечания про UDP

Мы должны отметить, что UDP не устанавливает соединения между клиентом и сервером, сервер не имеет методов блокировать клиента, отвергая чтение данных — сервер не имеет представления, кто является клиентом.

Большие UDP пакеты могут фрагментироваться, по мере прохождения через сеть. Фрагментация имеет место, когда размер UDP данных превышает величину maximum transfer unit (MTU), эта величина определяется узлами сети, через которые проходит пакет. Обычно, при настройке сети, рекомендуют на начальном этапе выставить MTU около 500 байтов, и затем постепенно увеличивать с измерение пропускной способности сети. Если на некотором узле пропускная способность резко упадет, тогда вы узнаете, что пакет является очень большим.

UDP пакеты могут быть доставлены дважды(что удивит некоторых людей), таким образом вы должны быть осторожны, когда пишете код для удаленных процедурных вызовов. Это может произойти, если ответ сервера на второй запрос совпадает с ответом сервера на первый запрос. Что бы избежать этого мы должны модифицировать клиентский код, включением уникальных идентификаторов, и проверкой, что сервер возвращает этот идентификатор. Для генерации уникальных идентификаторов, мы вызываем BIF make_ref, которая гарантирует возвращение глобального уникального идентификатора. Код для удаленного процедурного вызова теперь выглядит так:

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    Ref = make_ref(), %% создание уникального идентификатора
    B1 = term_to_binary({Ref, Request}),
    ok = gen_udp:send(Socket, "localhost" , 4000, B1),
    wait_for_ref(Socket, Ref).

wait_for_ref(Socket, Ref) ->
    receive
        {udp, Socket, _, _, Bin} ->
            case binary_to_term(Bin) of
                {Ref, Val} ->
                    %% получено корректное число
                    Val;
                {_SomeOtherRef, _} ->
                    %% пришло какое-то другое число. отбрасываем его.
                    wait_for_ref(Socket, Ref)
            end;
        after 1000 ->
            ...
    end.
```

14.6 Широковещание на множество машин

Для полноты картины, я покажу вам, как создавать широковещательный канал. Этот код вряд ли вам пригодится, но возможно, что в один день вам он понадобится.

Загрузить [broadcast.erl](#)

```
-module(broadcast).
-compile(export_all).

send(IoList) ->
    case inet:ifget("eth0", [broadaddr]) of
        {ok, [{broadaddr, Ip}]} ->
            {ok, S} = gen_udp:open(5010, [{broadcast, true}]),
            gen_udp:send(S, Ip, 6000, IoList),
            gen_udp:close(S);
        _ ->
            io:format("Bad interface name, or\n"
                      "broadcasting not supported\n")
    end.

listen() ->
    {ok, _} = gen_udp:open(6000),
    loop().

loop() ->
    receive
        Any ->
            io:format("received:~p~n", [Any]),
            loop()
    end.
```

Здесь нам понадобится два порта, один будет широковещать, а остальные прослушивать. Мы выбрали 5010 порт для отправки широковещательных запросов и 6000 для прослушивания широковещательного траффика(Эти два числа не имеют значения; я просто выбрал два свободных порта на моей системе).

Открытие 5010 порта производит только тот процесс, который производит широковещательную рассылку, а все остальные машины в сети вызывают `broadcast:listen()`, для открытия 6000 порта и прослушивания широковещательных сообщений. `broadcast:send(IoList)` посылает всем машинам в локальной сети `IoList` широковещательным сообщением.

Примечание: что бы это заработало, имя сетевого интерфейса должно быть корректным, и в локальной сети должно поддерживаться широковещание. На моем iMac, например, я использую имя "en0" вместо "eth0". Заметим так же, что если хост,

который производит широковещание и хост, который прослушивает UDP широковещательный трафик находятся в разных подсетях, то второй вряд ли услышит первого, поскольку по умолчанию маршрутизаторы не пропускают широковещательные сообщения за пределы подсети.

14.7 SHOUTcast сервер

В завершении этого раздела, мы используем новоприобретенные навыки в сокет-программировании для написания SHOUTcast сервера. SHOUTcast — это протокол, который разработали в Nullsoft, для потокового вещания аудио данных(6). SHOUTcast отправляет MP3 или AAC закодированные аудиоданные, используя HTTP, как транспортный протокол. Что бы увидеть, как это работает, вначале мы посмотрим SHOUTcast протокол. Затем мы посмотрим на общие структуры серверов, и закончим написанием кода.

14.7.1 SHOUTcast протокол

протокол SHOUTcast довольно прост:

1. Вначале клиент(который может быть чем-то вроде XMMS, Winamp или iTunes) посыпает HTTP запрос SHOUTcast серверу. Вот запрос, который XMMS генерирует, когда я запускаю мой SHOUTcast сервер дома:

```
GET / HTTP/1.1
Host: localhost
User-Agent: xmms/1.2.10
Icy-MetaData:1
```

2. Мой SHOUTcast сервер отвечает вот так:

```
ICY 200 OK
icy-notice1: <BR>This stream requires
<a href="http://www.winamp.com/">;Winamp</a><BR>
icy-notice2: Erlang Shoutcast server<BR>
icy-name: Erlang mix
icy-genre: Pop Top 40 Dance Rock
icy-url: http://localhost:3000
content-type: audio/mpeg
icy-pub: 1
icy-metaint: 24576
icy-br: 96
... data ...
```

3. Теперь SHOUTcast сервер посыпает непрерывный поток данных. данные имеют следующую структуру:

```
H0 F H F H F H ... (7)
```

`F` — это блок MP3 аудио данных, который должен быть ровно 24 576 байтов(число получено из `icy-metaint` параметра).

`H` — это блок метаданных. Первый байт блока `H` есть целое число `K`, `16*K` — это длина блока метаданных(без учета первого байта). Блок метаданных содержит строку вида: `StreamTitle=' ... ';` `SreamUrl=' ... ';`. Если длина этой строки не кратна `16`, то оставшиеся байты будут забиты нулями.

Заметим так же, что наикратчайший блок метаданных выглядит так `<<0>>`. То есть один байт длины с нулями.

(6) <http://www.shoutcast.com/>

(7) От автора перевода: В оригинальном тексте не упоминается про блок `H0`(хотя в коде он есть), но для лучшего восприятия я решил его написать. этот блок генерируется единожды, когда клиент отправляет запрос на получение данных. `H0` генерируется функцией `responce()`.

14.7.2 Как SHOUTcast сервер работает

Для создания сервера, мы будем соблюдать следующие детали:

1. Сделаем плейлист. Наш сервер будет использовать файл с ID3 тэгами, который мы создали в Главе 13.2. Песни будут выбираться случайным образом.
2. Сделаем параллельный сервер, таким образом мы сможем обслуживать несколько потоков параллельно. Мы сделаем это, используя технику, которая описана в 14.1(Параллельный сервер, на странице 254).
3. Из каждого файла мы будем отправлять только аудиоданные, без ID3 тэгов(8). Для удаления ID3 тэгов, мы будем использовать код из `id3_tag_length`; этот код использует код, разработанный на странице 232, код из раздела 5.3, Поиск и синхронизация фреймов в MPEG данных, на странице 92. Этот код не будет показан здесь.

(8) Непонятно, является ли это правильной стратегией. Аудио кодировщики предполагают перескакивание очень плохих данных, поэтому, мы можем отправлять ID3 тэги вместе с аудио-данными. Но на деле, кажется, что программа работает лучше, если мы удалим ID3 тэги.

14.7.3 Псевдокод для SHOUTcast сервера

Перед тем, как мы увидим финальную программу, давайте взглянем на общий вид программы, опустив некоторые детали:

```
start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, ..),
    %% создание музыкального сервера -- он просто знает обо всей нашей музь
    spawn(fun() -> par_connect(Listen, PidSongServer) end).

    %% порождение одного из процессов на ожидание соединения
    par_connect(Listen, PidSongServer) ->
        {ok, Socket} = gen_tcp:accept(Listen),
        %% когда accept возвращает управление, мы порождаем
        %% новый процесс для ожидания соединения.
        spawn(fun() -> par_connect(Listen, PidSongServer) end),
        inet:setopts(Socket, [{packet,0},binary, {nodelay,true},
            {active, true}]),
        %% deal with the request
        get_request(Socket, PidSongServer, []).

%% ожидает TCP соединение
get_request(Socket, PidSongServer, L) ->
    receive
        {tcp, Socket, Bin} ->
            ... Bin содержит клиентский запрос
            ... если запрос фрагментирован, то мы вызываем loop снова ...
            ... иначе мы вызываем
            ... got_request(Data, Socket, PidSongServer)
        {tcp_closed, Socket} ->
            ... это происходит, если клиент обрывает соединение
            ... до того, как успел послать запрос (маловероятно)
    end.

%% мы получили запрос --отправим ответ
got_request(Data, Socket, PidSongServer) ->
    .. data -это клиентский запрос ...
    .. анализируем его ...
    .. мы будем всегда обслуживать запрос ..
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer).

%% будем проигрывать песни, пока клиент не отсоединится.
play_songs(Socket, PidSongServer) ->
    ... PidSongServer хранит список всех MP3 данных
```

```

Song = rpc(PidSongServer, random_song),
... Song -это случайная песня ...
Header = make_header(Song),
... создание блока метаданных ...
{ok, S} = file:open(File, [read,binary,raw]),
send_file(1, S, Header, 1, Socket),
file:close(S),
play_songs(Socket, PidSongServer).

send_file(K, S, Header, OffSet, Socket) ->
    ... отправка файла клиенту фрагментами ...
    ... эта функция возвращает управление, когда файл отправлен ...
    ... если произошла ошибка, при записи в сокет
    ... это означает, что клиент отсоединился.

```

Если вы посмотрите на реальный код, то вы увидите, что детали слегка отличаются, но идея так же самая. Вот полный листинг:

[Загрузить shout.erl](#)

```

-module(shout).

%% в одном окне > shout:start()
%% в других окнах xmms http://localhost:3000/stream

-export([start/0]).
-import(lists, [map/2, reverse/1]).

-define(CHUNKSIZE, 24576).

start() ->
    spawn(fun() ->
        start_parallel_server(3000),
        %% теперь усыпляемся, поскольку если это не сделать,
        %% то прослушиваемый сокет будет закрыт.
        lib_misc:sleep(infinity)
    end).

start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, [binary, {packet, 0},
                                         {reuseaddr, true},
                                         {active, true}]),
    PidSongServer = spawn(fun() -> songs() end),
    spawn(fun() -> par_connect(Listen, PidSongServer) end).

```

```

par_connect(Listen, PidSongServer) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},{active, true}])
    get_request(Socket, PidSongServer, [])�.

get_request(Socket, PidSongServer, L) ->
    receive
        {tcp, Socket, Bin} ->
            L1 = L ++ binary_to_list(Bin),
            %% split checks if the header is complete
            case split(L1, []) of
                more ->
                    %% заголовок собран не полностью, нужно больше данных.
                    get_request(Socket, PidSongServer, L1);
                {Request, _Rest} ->
                    %% заголовок собран полностью
                    got_request_from_client(Request, Socket, PidSongServer)
            end;
        {tcp_closed, Socket} ->
            void;
        _Any ->
            %% пропустим это
            get_request(Socket, PidSongServer, L)
    end.

split("\r\n\r\n" ++ T, L)    -> {reverse(L), T};
split([H|T], L)             -> split(T, [H|L]);
split([], _)                -> more.

got_request_from_client(Request, Socket, PidSongServer) ->
    Cmds = string:tokens(Request, "\r\n"),
    Cmds1 = map(fun(I) -> string:tokens(I, " ") end, Cmds),
    is_request_for_stream(Cmds1),
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer, <>>).

play_songs(Socket, PidSongServer, SoFar) ->
    Song = rpc(PidSongServer, random_song),
    {File,PrintStr,Header} = unpack_song_descriptor(Song),
    case id3_tag_lengths:file(File) of
        error ->
            play_songs(Socket, PidSongServer, SoFar);
        {Start, Stop} ->

```

```

        io:format("Playing:~p~n" ,[PrintStr]),
        {ok, S} = file:open(File, [read,binary,raw]),
        SoFar1 = send_file(S, {0,Header}, Start, Stop, Socket, SoFar),
        file:close(S),
        play_songs(Socket, PidSongServer, SoFar1)
    end.

send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
    %% OffSet = первый байт аудиоданных.
    %% Stop = последний байт аудиоданных.
    Need = ?CHUNKSIZE -size(SoFar),
    Last = OffSet + Need,
    if
        Last >= Stop ->
            %% даже если мы и дочитаем файл до конца, то мы не
            %% наберем байтов до 24576, поэтому вычитываем, что есть
            %% и возвращаемся в play_songs
            Max = Stop - OffSet,
            {ok, Bin} = file:pread(S, OffSet, Max),
            list_to_binary([SoFar, Bin]);
        true ->
            {ok, Bin} = file:pread(S, OffSet, Need),
            write_data(Socket, SoFar, Bin, Header),
            send_file(S, bump(Header),
                      OffSet + Need, Stop, Socket, <>>)
    end.

write_data(Socket, B0, B1, Header) ->
    %% Проверим, действительно ли данные для отправки имеют нужную длину.
    %% это очень полезная проверка, которая проверяет нашу программу на корректность.
    %% case size(B0) + size(B1) of
    %%     ?CHUNKSIZE ->
    %%         case gen_tcp:send(Socket, [B0, B1, the_header(Header)]) of
    %%             ok -> true;
    %%             {error, closed} ->
    %%                 %% это происходит, если медиаплеер
    %%                 %% прерывает соединение.
    %%                 exit(playerClosed)
    %%         end;
    %%     _Other ->
    %%         %% не посылаем блок, а сигнализируем об ошибке.
    %%         io:format("Block length Error: B0 = ~p b1=~p~n" ,
    %%                  [size(B0), size(B1)])
end.

```

```

bump({K, H}) -> {K+1, H}.

the_header({K, H}) ->
    case K rem 5 of
        0 -> H;
        _ -> <<0>>
    end.

is_request_for_stream(_) -> true.

response() ->
    ["ICY 200 OK\r\n" ,
     "icy-notice1: <BR>This stream requires" ,
     "<a href=\" http://www.winamp.com/\">Winamp</a><BR>\r\n" ,
     "icy-notice2: Erlang Shoutcast server<BR>\r\n" ,
     "icy-name: Erlang mix\r\n" ,
     "icy-genre: Pop Top 40 Dance Rock\r\n" ,
     "icy-url: http://localhost:3000\r\n" ,
     "content-type: audio/mpeg\r\n" ,
     "icy-pub: 1\r\n" ,
     "icy-metaint: " ,integer_to_list(?CHUNKSIZE),"\r\n" ,
     "icy-br: 96\r\n\r\n"].

songs() ->
    {ok,[SongList]} = file:consult("mp3data" ),
    lib_misc:random_seed(),
    songs_loop(SongList).

songs_loop(SongList) ->
    receive
        {From, random_song} ->
            I = random:uniform(length(SongList)),
            Song = lists:nth(I, SongList),
            From ! {self(), Song},
            songs_loop(SongList)
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} ->
            Reply
    end.

unpack_song_descriptor({File, {_Tag,Info}}) ->

```

```

PrintStr = list_to_binary(make_header1(Info)),
L1 = ["StreamTitle='", PrintStr,
      "' ; StreamUrl='http://localhost:3000';" ],
%% io:format("L1=~p~n",[L1]),
Bin = list_to_binary(L1),
Nblocks = ((size(Bin) -1) div 16) + 1,
NPad = Nblocks*16 -size(Bin),
Extra = lists:duplicate(NPad, 0),
Header = list_to_binary([Nblocks, Bin, Extra]),
%% Header это блок метаданных.
{File, PrintStr, Header}.

make_header1([{track,_}|T]) ->
    make_header1(T);
make_header1([{Tag,X}|T]) ->
    [atom_to_list(Tag),": ", X, " " |make_header1(T)];
make_header1([]) ->
    [].

```

14.7.4 Запустим SHOUTcast сервер

Запустим сервер, и проверим, как он работает, нам необходимо выполнить три шага:

1. Создать плейлист.
2. Запустить сервер.
3. Настроить клиент для работы с сервером.

14.7.5 Создание плейлиста

Для создания плейлиста нам надо выполнить следующие шаги:

1. Перейти в каталог, где лежит модуль `mp3_manager.erl` (9)
2. Изменить путь в функции `start1`, которая находится в файле `mp3_manager.erl`, на путь, который указывает на каталог с MP3 файлами.
3. Скомпилировать `mp3_manager`, и набрать в оболочке `mp3_manager:start1()`. Мы должны увидеть что-то вроде этого:

```

1> c(mp3_manager).
{ok,mp3_manager}
2> mp3_manager:start1().
Dumping term to mp3data

```

ok

Если вам интересно, то вы можете взглянуть на файл `mp3data`, чтобы увидеть результаты анализа.

14.7.6 Запуск SHOUTcast сервера

Запустим сервер из оболочки следующей командой:

```
1> shout:start().  
...
```

14.7.7 Тестирование сервера

1. Запустим плеер и укажем в его настройках адрес сервера:

`http://localhost:3000` На моей системе я использовал XMMS, которого запустил следующей командой: `xmms http://localhost:3000`

Примечание: если вы хотите подключиться к серверу с другой машины, вы должны указать IP адрес сервера. Например, что бы подключиться к серверу с моей Windows машины, на которой установлен winamp, я вызвал Play > URL меню в винампе и ввел адрес `http://192.168.1.168:3000` в диалоговом окне Open URL на моем iMac я использовал iTunes, я вызвал Advanced > Open Stream меню и вписал в него предыдущий url.

2. Вы увидите диагностический вывод в окне, в котором запущен сервер.
3. Enjoy!

(9) тот самый модуль из предыдущей главы, где мы извлекали метаданные из MP3 файлов.

14.8 Копаем глубже

В этой главе мы рассмотрели наиболее часто используемые функции для манипулирования сокетами. Вы можете найти больше информации о socket API в документации, на страницах `gen_tcp`, `gen_udp`, и `inet`.

Глава 15. ETS и DETS: механизмы хранения данных

`ets` и `dets` — это два системных модуля, используемых для эффективного хранения

большого количества данных. `ETS` является сокращением от Erlang term storage (хранилище данных Эрланга), а `DETS` — сокращением от disk Erlang term Storage (дисковое хранилище данных эрланга).

`ETS` и `DETS` выполняют одинаковую задачу: предоставляют большие таблицы для хранения данных в формате ключ-значение. `ETS` хранит данные в памяти, а `DETS` на диске. Механизм `ETS` очень эффективен — используя `ETS`, вы можете хранить огромные количества данных (если у вас достаточно памяти) и осуществлять выборки за постоянное время (или, в некоторых случаях, за логарифмическое). `DETS` предоставляет интерфейс похожий на интерфейс `ETS`, но хранит таблицы на диске. Так как `DETS` хранит данные на диске, он гораздо медленнее чем `ETS`, но использует гораздо меньше памяти во время работы. Несколько процессов могут совместно работать с `ETS` и `DETS` таблицами, при этом обеспечивается высокоэффективный доступ к общим данным.

`ETS` и `DETS` представляют собой структуры данных для задания соответствия между ключами и значениями. Мы рассмотрим наиболее общие операции над таблицами: вставку и выборку. `ETS` и `DETS` таблицы представляют собой просто список кортежей Эрланга.

Данные в `ETS` являются временными и будут удалены, когда завершатся процессы работающие с таблицей. Данные же хранимые в `DETS` являются постоянными и должны остаться даже в случае краха всей системы. При открытии `DETS` таблица проверяется на консистентность. Если она повреждена, то предпринимается попытка восстановить таблицу (что может занять долгое время, пока все данные в таблице будут проверены). Это должно восстановить все данные в таблице, хотя последняя запись в таблице может потеряться, если она попала на момент краха системы.

`ETS` таблицы широко используются для программ, которые эффективно работают с большими количествами данных, и где слишком дорого использовать неразрушающее присваивание и базовые структуры данных Эрланга.

`ETS` таблицы выглядят как если бы они были реализованы на Эрланге, но на самом деле они реализованы на более низком системном уровне и обладают производительностью отличной от обычных объектов Эрланга. В частности, `ETS` таблицы не обрабатываются сборщиком мусора; это означает, что сборщик мусора не мешает при работе с очень большими таблицами, хотя он может немного повлиять на операции создания или доступа к `ETS` объектам.

15.1 Базовые операции с таблицами

Существуют четыре базовые операции с ETS и DETS таблицами:

Создание новой таблицы или открытие существующей.

Это производится с помощью `ets:new` или `dets:open_file`.

Вставка одного или нескольких кортежей в таблицу.

Вызываем `insert(Tablename, X)`, где `X` — кортеж, или список кортежей. В ETS и DETS функция `insert` принимает одинаковые аргументы и выполняет одинаковую операцию.

Поиск кортежа в таблице.

Вызываем `lookup(TableName, Key)`. Результатом будет список кортежей, которые соответствуют ключу `Key`. `lookup` определена для ETS и DETS.

(Почему возвращаемое значение список кортежей? Если тип таблицы "bag", то одному значению ключа может соответствовать несколько кортежей. Мы рассмотрим типы таблиц в следующем параграфе.)

Если ни один из кортежей не соответствует указанному ключу, вернётся пустой список.

Завершение работы с таблицей.

Когда мы закончили работать с таблицей, мы можем сообщить об этом системе вызвав `dets:close(TableId)` или `ets:delete(TableId)`.

15.2 Типы таблиц

ETS и DETS таблицы хранят кортежи. Один из элементов кортежа (по умолчанию — первый) называется ключом таблицы. Мы вставляем данные в таблицу и извлекаем данные оттуда по ключу. Что происходит, когда мы вставляем в таблицу данные, зависит от типа таблицы и значения вставляемого ключа. Некоторые таблицы, называемые множествами `set`, требуют, чтобы все ключи в таблице были уникальными. Другие, называемые сумками `bag`, позволяют нескольким кортежам иметь одинаковый ключ.

Выбор правильного типа таблицы оказывает сильное влияние на производительность ваших программ.

Каждый из двух типов таблиц имеет два варианта, что порождает четыре типа таблиц: множество (`set`), упорядоченное множество (`ordered set`), сумка (`bag`) и сумка с

повторами (duplicate bag). В множестве все ключи кортежей таблицы должны быть уникальными. В упорядоченном множестве кортежи отсортированы. В сумке может быть более одного кортежа с одним ключом, но не может быть двух полностью одинаковых кортежей. В сумке с повторами несколько кортежей могут обладать одинаковым ключом, и одинаковые кортежи могут встречаться много раз.

Мы можем показать как это работает с помощью следующей маленькой программы:

[Скачать ets_test.erl](#)

```
-module(ets_test).
-export([start/0]).

start() ->
    lists:foreach(fun test_ets/1,
        [set, ordered_set, bag, duplicate_bag]).

test_ets(Mode) ->
    TableId = ets:new(test, [Mode]),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {b,2}),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {a,3}),
    List = ets:tab2list(TableId),
    io:format("\~-13w => \~p\~n", [Mode, List]),
    ets:delete(TableId).
```

Эта программа открывает ETS таблицу в каждом из четырёх режимов и вставляет кортежи `{a,1}`, `{b,2}`, `{a,1}` и `{a,3}`. Тогда мы вызываем `tab2list`, которая преобразует таблицу целиком в список, и печатаем её.

Когда мы запустим программу, получим:

```
1> ets_test:start().
set => [{b,2},{a,3}]
ordered_set => [{a,3},{b,2}]
bag => [{b,2},{a,1},{a,3}]
duplicate_bag => [{b,2},{a,1},{a,1},{a,3}]
```

Для множества `set` каждый ключ встречается только один раз. Если мы вставили кортеж `{a,1}`, а вслед за ним `{a,3}`, тогда итоговое значение будет `{a,3}`. Единственное отличие между упорядоченным множеством (ordered set) и множеством состоит в том, что элементы упорядочены по значению ключа. Мы можем увидеть порядок, когда преобразуем таблицу в список вызвав `tab2list`.

Сумочные типы таблиц могут содержать несколько записей с одним ключом. Так, например, когда мы вставляем `{a, 1}` и `{a, 3}`, сумка (bag) будет содержать оба кортежа, а не только последний. В сумке с повторами (duplicate bag) допустимо наличие нескольких идентичных кортежей. Так, когда мы вставляем `{a, 1}` и `{a, 1}` в сумку с повтором, результирующая таблица содержит две копии кортежа `{a, 1}`, хотя в обычной сумке, была бы только одна копия кортежа.

15.3 Соображения об эффективности ETS таблиц

Внутри, ETS таблицы представляются хеш таблицами (за исключением упорядоченных множеств, которые представляются сбалансированными двоичными деревьями). Это означает, что происходят незначительные затраты на память при использовании множеств и временные затраты при использовании упорядоченных множеств. Вставка в множество занимает постоянное время, но вставка в упорядоченное множество занимает время пропорциональное логарифму общего количества записей в таблице.

Когда вы выбираете между множеством и упорядоченным множеством, вы должны решить, что будете делать с таблицей после её создания — если вам нужна отсортированная таблица, используйте упорядоченное множество.

Сумки более накладно использовать чем сумки с повторами, так как на каждую вставку все элементы с заданным ключом будут сравниваться на полную идентичность. Если число элементов с одинаковым ключом велико, то использование простых сумок может быть очень неэффективным.

ETS таблицы располагаются в отдельных областях памяти, которые не связаны с обычной памятью процесса. ETS таблица привязывается к процессу, который создал её — когда этот процесс умирает, или когда вызывается `ets:delete`, таблица удаляется. ETS таблицы не обрабатываются сборщиком мусора, это означает, что большие объёмы данных могут храниться в таблице без затрат на сборку мусора.

Когда кортеж вставляется в ETS таблицу, все структуры данных этого кортежа копируются из стека процесса и кучи в ETS таблицу. Когда производится выборка, результирующие кортежи копируются из ETS таблицы в стек и кучу процесса.

Сказанное верно для всех структур за исключением двоичных данных. Большие двоичные данные располагаются в отдельных хранилищах (не в куче). Эти хранилища могут совместно использоваться несколькими процессами и ETS таблицами. Двоичные данные управляются сборщиком мусора основанным на счётчике ссылок, который отслеживает сколько различных процессов и ETS таблиц использует двоичные данные. Когда счётчик ссылок опускается до нуля, место в хранилище может быть

освобождено.

Всё это может показаться довольно сложным, но в результате отправка сообщений с большими двоичными данными между процессами довольно быстра, и вставка кортежей в ETS таблицы, которые содержат большие двоичные данные также очень быстра. Хорошее правило использовать двоичные данные как можно чаще для отображения строк и больших блоков нетипизированных данных.

15.4 Создание ETS таблицы

ETS таблицы создаются вызовом `ets:new`. Процесс, который создаёт таблицу называется "владельцем" (owner) таблицы. При создании таблицы, у неё есть набор параметров, которые не могут быть изменены в дальнейшем. Если процесс владелец умирает, место из под таблицы автоматически освобождается; либо таблица может быть удалена вызовом `ets:delete`.

`ets:new` принимает следующие аргументы:

```
@spec ets:new(Name, [Opt]) -> TableId
```

`Name` - атом. `[Opt]` - список параметров из следующего перечня:

`set | ordered_set | bag | duplicate_bag` - Создать ETS таблицу указанного типа (мы говорили о них ранее).

`private` - Создать скрытую таблицу. Только процесс владелец может читать и писать в эту таблицу.

`public` - Создать публичную таблицу. Любой процесс, который знает идентификатор таблицы, может читать и писать в таблицу.

`protected` - Создать защищённую таблицу. Любой процесс, который знает идентификатор таблицы, может читать из неё, но только владелец может писать.

`named_table` - Если указан, то Name может быть использован для последовательных операций с таблицами.

`{keypos, K}` - Использовать K как позицию ключа в кортеже. Обычно позиция 1 используется для ключа. Вероятно, единственный случай, когда нам понадобится использовать этот параметр, если мы храним Erlang record,(который на самом деле замаскированный кортеж), где первый элемент содержит имя record.

ETS таблицы как аудиторные доски

Защищённые таблицы предоставляют некоторое подобие аудиторных досок (доски для мела в аудиториях). Вы можете думать о ETS таблицах как о поименованных аудиторных досках. Каждый, кто знает имя доски, может читать с ней, но только владелец может писать на ней.

Замечание: Если ETS таблица была открыта в публичном режиме, то любой процесс, который знает её имя может и читать и писать в неё. В этом случае, пользователь должен самостоятельно убедиться, что чтение и запись производятся правильно и не нарушают целостности данных.

Замечание: Открытие таблицы ETS с пустыми параметрами равнозначно открытию таблицы с параметрами `[set, protected, {keypos, 1}]`.

Весь код в этой главе использует защищённые ETS таблицы. Защищённые таблицы особенно полезны, так как они позволяют предоставить общий доступ к данным практически без накладных затрат. Все локальные процессы, которые знают идентификатор таблицы, могут читать данные, но только один процесс может изменять таблицу.

15.5 Примеры программ использующих ETS

Примеры в данном параграфе предназначены для работы с формированием триграмм. Это отличный способ продемонстрировать силу ETS таблиц.

Наша цель — написать эвристическую программу, которая пытается предсказать, является ли заданная строка английским словом. Мы собираемся использовать её, когда будем создавать модуль полнотекстового поиска в параграфе 20.4, mapreduce и проводить индексирование нашего диска на странице 379.

Как же мы можем предсказать, является ли случайная последовательность букв действительным английским словом? Один из способов — использовать триграммы. Триграмма - это последовательность трёх букв. Не все триграммы допустимы в действительном английском слове. Например, не существует английских слов, где существуют комбинации akj или rwb. Итак, для проверки, может ли быть заданная строка английским словом, мы будем проверять все наборы из трёх последовательных букв в строке на предмет совпадения с множеством триграмм, собранном на большом количестве английских слов.

Первое, что наша программа сделает, это вычислит все триграммы в английском языке на очень большом множестве слов. Чтобы сделать это, мы воспользуемся таблицей ETS с типом множество. Решение использовать ETS множества основано на результатах измерения относительной производительности множеств и

упорядоченных множеств ETS, а также "чистых" Эрланговых множеств предоставляемых модулем `set`.

Вот что мы собираемся сделать:

1. Создать итератор, который проходит по всем триграммам английского языка. Это очень сильно упростит написание кода вставки триграмм в различные типы таблиц.
2. Создадим ETS таблицы с типами множество и упорядоченное множество для хранения всех триграмм. Также, создадим множество содержащее все эти триграммы.
3. Замерим сколько времени потребуется для построения таблиц.
4. Замерим время поиска в этих таблицах.
5. Основываясь на измерениях, выберем наилучший тип и напишем функции доступа для этого типа.

Весь код находится в `lib_trigrams`. Мы собираемся представить его упустив некоторые детали. Но не беспокойтесь, вы найдёте полный листинг программы в конце главы. Таков наш план. Итак, приступим.

Итератор триграмм

Мы определим функцию `for_each_trigram_in_the_english_language(F, A)`. Эта функция применяет `fun F` к каждой триграмме в английском языке. `F` - это `fun` типа `fun(Str, A) -> A`, а `Str` пробегает все значения триграмм в языке и `A` - это аккумулятор.

Чтобы написать наш итератор^[^1], нам потребуется большой список слов. Я использовал набор из 354984 английских слов^[^2], для формирования триграмм. Используя этот список слов, мы можем определить итератор триграмм следующим способом:

Скачать [lib_trigrams.erl](#)

```
for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).
    scan_word_list([], _, A) ->
        A;
```

```

scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([$s|Word], F, A),
    scan_word_list(L1, F, A1).

%% scan the word looking for \r\n
%% the second argument is the word (reversed) so it
%% has to be reversed when we find \r\n or run out of characters

get_next_word([$r,$\n|T], L)    -> {reverse([$s|L]), T};
get_next_word([H|T], L)         -> get_next_word(T, [H|L]);
get_next_word([], L)           -> {reverse([$s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);
scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.

```

Здесь следует отметить два момента. Во-первых, мы использовали `zlib:gunzip(Bin)` чтобы разархивировать zip-архив. Список слов достаточно велик, так что мы предпочли сохранить его на диск в сжатом виде, а не как текстовый файл. Во-вторых, мы добавили пробел до и после каждого слова `Second`; в нашем анализаторе триграмм, мы хотим интерпретировать пробел как обычную букву.

Строим таблицы

Мы строим наши ETS таблицы так:

[Скачать lib_trigrams.erl](#)

```

make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab").
make_ets_set()           -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.

```

Обратите внимание, как мы вставляли отдельные триграмммы, например ABC. Мы на самом деле мы вставляли кортеж `{<<"ABC">>}` в таблицу ETS. Это выглядит забавно — кортеж только с одним элементом. Что это значит? Конечно, кортеж это контейнер для нескольких элементов, но это не запрещает иметь контейнер только с одним элементом. Но вспомните, что все записи в ETS таблицах должны быть кортежами, и по умолчанию первый элемент кортежа является ключом. Итак, в нашем случае, кортеж `{Key}` является ключом без значения.

Вот код, который строит множество всех триграмм (на сей раз с использованием Эрлангового модуля sets и без ETS):

Скачать [lib_trigrams.erl](#)

```
make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) -> sets:add_element(list_to_binary(Str), Set) end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).
```

Сколько времени занимает построение таблиц?

Функция `lib_trigrams:make_tables()`, показанная в листинге в конце главы, строит все таблицы. Она включает несколько инструкций, которые помогут определить размер таблиц и время необходимое для их создания.

```
1> lib_trigrams:make_tables().
Counting - No of trigrams=3357707 time/trigram=0.577938
Ets ordered Set size=19.0200 time/trigram=2.98026
Ets set size=19.0193 time/trigram=1.53711
Module Set size=9.43407 time/trigram=9.32234
ok
```

О чём это говорит нам? Во-первых, у нас 3.3 миллиона триграмм, и требуется пол микросекунды для обработки каждой триграмммы. Время вставки триграмммы в ETS упорядоченное множество было 2.9 микросекунды, в ETS множество было 1.5 микросекунды и 9.3 микросекунды в множество Эрланга. Что касается хранилищ, ETS множества и упорядоченные множества потребляют 19 байт на триграммму, а модуль sets потребляет 9 байт на триграммму.

Каково время доступа к таблицам?

Итак, требуется некоторое время для построения таблиц, но в нашем случае это не так уж важно. Более важный вопрос, каково время доступа к таблице? Чтобы ответить на этот вопрос, надо написать код, который будет измерять время доступа к таблице.

Мы будем искать каждую триграмму в таблице ровно один раз и определять среднее время на поиск. Вот код, который определяет время:

[Скачать lib_trigrams.erl](#)

```
timer_tests() ->
    time_lookup_ets_set("Ets ordered Set" , "trigramsOS.tab" ),
    time_lookup_ets_set("Ets set" , "trigramsS.tab" ),
    time_lookup_module_sets().

time_lookup_ets_set(Type, File) ->
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("\~s lookup=\~p micro seconds\~n" ,[Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set" ),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=\~p micro seconds\~n" ,[M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).
```

Вот что мы получим:

```
1> lib_trigrams:timer_tests().
Ets ordered Set lookup=1.79964 micro seconds
Ets set lookup=0.719279 micro seconds
Module sets lookup=1.35268 micro seconds
ok
```

Здесь время - среднее время за одну выборку.

И победитель это...

Ну, здесь всё просто. ETS множество выиграло с большим отрывом. На моей машине, множеству требуется полмикросекунды на выборку - это очень хорошо!

Примечание: выполнение тестов наподобие этих, и вообще измерения сколько требуется конкретной операции, является признаком хорошего стиля программирования. Нам не надо моделировать экстремальные условия и измерять всё подряд, только наиболее долгие операции в нашей программе. Быстрые операции должны быть запрограммированы наиболее красивым способом. Если нам приходится писать невероятно страшный и запутанный код, надо его детально документировать.

А сейчас мы можем написать функцию, которая попробует предсказать, является ли строка действительным английским словом.

Чтобы определить является ли строка английским словом, мы просматриваем все триграммы в строке и проверяем каждую на наличие в вычисленном ранее в списке. Функция `is_word` делает это.

[Скачать lib_trigrams.erl](#)

```
is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").

is_word1(Tab, [_,_,_] = X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true -> is_word1(Tab, [B,C|D]);
        false -> false
    end;
is_word1(_, _) ->
    false.

is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _ -> true
    end.

open() ->
{ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE))
                      ++ "/trigramsS.tab"),
I.

close(Tab) -> ets:delete(Tab).
```

Функции `open` и `close` открывают ETS таблицу и заполняют её. Всякий вызов `is_word` должен "обращиваться" в эти функции.

Другой трюк, который я использовал здесь — метод, которым я определяю внешний файл с таблицей триграмм. Я положил его в директорию из которой был загружен

текущий модуль. `code:which(?MODULE)` возвращает имя файла, в котором обнаружен объектный код для `?MODULE`.

15.6 DETS

DETS предоставляет хранилище кортежей на диске. Максимальный размер DETS файла 2Гб. DETS файлы должны быть открыты перед использованием и правильно закрыты после. Если они не были правильно закрыты, они будут автоматически восстановлены при следующем открытии. Так как восстановление может занять долгое время, важно закрывать их правильно до завершения приложения.

У DETS таблиц параметры совместного доступа отличаются от ETS таблиц. Когда открывается DETS таблица, ей должно быть присвоено глобальное имя. Если два или более локальных процесса откроют DETS таблицу с одинаковым именем и параметрами, они будут иметь к ней совместный доступ. Таблица будет оставаться открытой до тех пор пока все процессы не закроют её (или пока они не завершатся).

Пример: Индекс имён файлов

Мы начнём с примера и ещё одной утилиты, которая нам потребуется для полнотекстового движка, который будет рассматриваться в параграфе 20.4, для mapreduce и параграфа "Индексируем наш диск", на странице 379.

Мы собираемся создать дисковую таблицу, которая поставит соответствие целых чисел именам файлов (пронумерует), и т.д. Мы определим функцию `filename2index` и обратную ей `index2filename`.

Чтобы реализовать их, мы создадим DETS таблицу и поместим в неё три различных типа кортежей:

`{free, N}` - `N` это первый свободный индекс в таблице. Когда мы вставляем новое имя файла в таблицу, ему будет присвоен индекс `N`.

`{FileNameBin, K}` - `FileNameBin` (двоичная строка) соответствует индексу `K`.

`{K, FileNameBin}` - `K` (целое) соответствует файлу `FileNameBin`.

Заметьте как добавление каждого нового файла, добавляет две новые строки в таблицу: запись Файл -> Индекс и запись Индекс -> Файл. Это сделано из соображений эффективности. Когда ETS или DETS таблицы создаются, только один элемент кортежа может выступать в роли ключа. Поиск кортежа не по ключевому полю возможен, но он очень неэффективен, потому что он приводит к перебору всех значений таблицы. Это достаточно дорогостоящая операция, особенно, когда таблица

располагается на диске.

Сейчас давайте напишем программу. Начнём с функций открытия и закрытия DETS таблицы для хранения имён всех наших файлов.

Скачать [lib_filenames_dets.erl](#)

```
-module(lib_filenames_dets).
-export([open/1, close/0, test/0, filename2index/1, index2filename/1]).

open(File) ->
    io:format("dets opened:~p~n" , [File]),
    Bool = filelib:is_file(File),
    case dets:open_file(?MODULE, [{file, File}]) of
        {ok, ?MODULE} ->
            case Bool of
                true -> void;
                false -> ok = dets:insert(?MODULE, {free,1})
            end,
            true;
        {error,_Reason} ->
            io:format("cannot open dets table~n" ),
            exit(eDetsOpen)
    end.

close() -> dets:close(?MODULE).
```

Код для открытия автоматически инициализирует DETS таблицу вставляя кортеж `{free, 1}`, если была создана новая таблица. `filelib:is_file(File)` возвращает `true`, если файл `File` существует, в противном случае, она возвращает `false`. Заметьте, что `dets:open_file` создаёт новый файл, или открывает существующий, именно поэтому мы проверяем существовал ли файл до вызова `dets:open_file`.

В этом коде я использовал макрос `?MODULE` много раз; `?MODULE` преобразуется в имя текущего модуля `lib_filenames_dets`. Многие вызовы DETS таблиц требуют уникальный атом — имя таблицы.

Для генерации уникального имени таблицы нам достаточно имени модуля. Так как в системе не может быть загружено два модуля с одним именем, то, следуя этому соглашению везде, мы обоснованно можем считать, что мы присваиваем уникальные имена таблицам.

Я использовал макрос `?MODULE` вместо того, чтобы каждый раз писать имя модуля, потому что у меня есть привычка менять названия модулей, пока я пишу код. Если использовать макрос, то даже при изменении имени модуля код останется верным.

Мы уже открыли файл, вставка нового имени файла в таблицу очень проста. Это реализовано как сторонний эффект вызова `filename2index`. Если имя файла уже есть в таблице, то возвращается его индекс; в противном случае рассчитывается новый индекс и таблица обновляется, на этот раз тремя кортежами:

Скачать [lib_filenames_dets.erl](#)

```
filename2index(FileName) when is_binary(FileName) ->
    case dets:lookup(?MODULE, FileName) of
        [] ->
            [{_,Free}] = dets:lookup(?MODULE, free),
            ok = dets:insert(?MODULE,
                [{Free,FileName},{FileName,Free},{free,Free+1}]),
            Free;
        [{_,N}] ->
            N
    end.
```

Заметьте как мы записываем три кортежа в таблицу. Второй аргумент `dets:insert` должен быть либо кортежем, либо списком кортежей. Заметьте также, что имя файла представлено двоичным значением. Это сделано из соображений эффективности. Вообще хорошо иметь привычку использовать двоичные данные для представления строк в ETS и DETS таблицах.

Внимательный читатель может заметить, что в примере есть возможный race condition (состояние гонки) в `filename2index`. Если два параллельных процесса вызовут `dets:lookup` до вызова `dets:insert`, в этом случае `filename2index` вернёт некорректное значение. Чтобы код работал, мы должны убедиться, что он работает в единственном экземпляре в каждый момент времени.

Преобразовать индекс в имя файла просто:

Скачать [lib_filenames_dets.erl](#)

```
index2filename(Index) when is_integer(Index) ->
    case dets:lookup(?MODULE, Index) of
        []      -> error;
        [{_,Bin}] -> Bin
    end.
```

Небольшое дизайнерское решение. Что будет произойдёт, если мы вызовем `index2filename(Index)`, а в таблице не будет имени файла привязанного к этому индексу? Мы можем завершиться вызвав `exit(ebadIndex)`. Но мы выбрали более элегантную альтернативу: мы просто возвращаем атом `error`. Вызывающий код

может сам выбрать между правильным и неправильным значениями, так как все правильные имена файлов имеют тип `binary`.

Отметьте также защитники (guard tests) в `filename2index` и `index2filename`. Они проверяют правильный ли тип имеют аргументы. Вообще, это правильно, потому что ввод данных неверного типа в DETS таблицу может вызвать ситуации, которые очень трудно отлаживать после. Мы можем представить запись данных с неверным типом в таблицу и чтение её через несколько месяцев, когда уже поздно делать что-либо. Потому, лучше проверять, что все данные верны перед вставкой в таблицу.

15.7 О чём мы не сказали?

ETS и DETS таблицы поддерживают набор операций, которые мы не рассмотрели в этой главе. Эти операции разделяются на следующие категории:

- получение и удаление объектов по шаблону
- преобразования между ETS и DETS таблицами и между ETS таблицами и файлами на диске
- определение ресурсов используемых таблицей
- пересечение (traversing) всех элементов в таблице
- восстановление повреждённых DETS таблиц
- визуализация таблиц

Вы можете найти больше информации на страницах руководства, доступного на <http://www.erlang.org/doc/man/ets.html> и <http://www.erlang.org/doc/man/dets.html>.

В заключение скажем, что ETS и DETS таблицы изначально разрабатывались для реализации Mnesia. Мы ещё не говорили о Mnesia, так как она — тема главы 17: База данных Эрланга, страница 313. Mnesia — база данных реального времени написанная на Эрланге. Mnesia использует ETS и DETS таблицы внутри себя, и много функций из модулей ETS и DETS предназначены для внутреннего пользования Mnesia. Mnesia может выполнять все типы операций, которые невозможно сделать на чистых ETS и DETS таблицах. Например, мы можем создавать индексы не только по первичному ключу, подобный пример мы использовали при двойной вставке в функции `filename2index`. Mnesia на самом деле создаёт несколько ETS или DETS таблиц, для реализации такого функционала, но это создание спрятано от пользователя.

15.8 Листинги кода

[Скачать lib_trigrams_complete.erl](#)

```
%%%--  
% Excerpted from "Programming Erlang",  
% published by The Pragmatic Bookshelf.  
% Copyrights apply to this code. It may not be used to create  
% training material,  
% courses, books, articles, and the like. Contact us if you  
% are in doubt.  
% We make no guarantees that this code is fit for any purpose.  
% Visit http://www.pragmaticprogrammer.com/titles/jaerlang for more  
% book information.  
%%%--  
  
-module(lib_trigrams).  
-export([for_each_trigram_in_the_english_language/2,  
        make_tables/0, timer_tests/0,  
        open/0, close/1, is_word/2,  
        how_many_trigrams/0,  
        make_ets_set/0, make_ets_ordered_set/0, make_mod_set/0,  
        lookup_all_ets/2, lookup_all_set/2  
        ]).  
-import(lists, [reverse/1]).  
  
make_tables() ->  
    {Micro1, N} = timer:tc(?MODULE, how_many_trigrams, []),  
    io:format("Counting - No of trigrams=~p time/trigram=~p~n",  
             [N, Micro1/N]),  
    {Micro2, Ntri} = timer:tc(?MODULE, make_ets_ordered_set, []),  
    FileSize1 = filelib:file_size("trigramsOS.tab"),  
    io:format("Ets ordered Set size=~p time/trigram=~p~n",  
             [FileSize1/Ntri, Micro2/N]),  
    {Micro3, _} = timer:tc(?MODULE, make_ets_set, []),  
    FileSize2 = filelib:file_size("trigramsS.tab"),  
    io:format("Ets set size=~p time/trigram=~p~n",  
             [FileSize2/Ntri, Micro3/N]),  
    {Micro4, _} = timer:tc(?MODULE, make_mod_set, []),  
    FileSize3 = filelib:file_size("trigrams.set"),  
    io:format("Module sets size=~p time/trigram=~p~n",  
             [FileSize3/Ntri, Micro4/N]).  
  
make_ets_ordered_set() ->  
    make_a_set(ordered_set, "trigramsOS.tab").  
make_ets_set() ->  
    make_a_set(set, "trigramsS.tab").
```

```

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.

make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) ->
        sets:add_element(list_to_binary(Str), Set)
    end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).

timer_tests() ->
    time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"),
    time_lookup_ets_set("Ets set", "trigramsS.tab"),
    time_lookup_module_sets().

time_lookup_ets_set(Type, File) ->
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("~s lookup=~p micro seconds~n", [Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set"),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=~p micro seconds~n", [M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).

```

```

how_many_trigrams() ->
    F = fun(_, N) -> 1 + N end,
    for_each_trigram_in_the_english_language(F, 0).

%% An iterator that iterates through all trigrams in the language

for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).

scan_word_list([], _, A) ->
    A;
scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([\$\\s|Word], F, A),
    scan_word_list(L1, F, A1).

%% scan the word looking for \\r\\n
%% the second argument is the word (reversed) so it
%% has to be reversed when we find \\r\\n or run out of characters

get_next_word([\$\$\r,\$\$\n|T], L)    -> {reverse([\$\$\s|L]), T};
get_next_word([H|T], L)              -> get_next_word(T, [H|L]);
get_next_word([], L)                -> {reverse([\$\$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);

scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.

%% access routines
%% open() -> Table
%% close(Table)
%% is_word(Table, String) -> Bool

is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").

```

```

is_word1(Tab, [_,_,_] = X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true   -> is_word1(Tab, [B,C|D]);
        false  -> false
    end;
is_word1(_, _) ->
    false.

is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _  -> true
    end.

open() ->
{ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE))
                        ++ "/trigramsS.tab"),
I.

close(Tab) -> ets:delete(Tab).

```

Глава 16. Введение в OTP

OTP означает Открытая Телекоммуникационная Платформа (Open Telecom Platform). На самом деле это название обманчиво, потому что OTP имеет более широкое применение, чем может показаться. OTP - это часть операционной системы с набором библиотек и процедур, используемых для построения масштабируемых, отказоустойчивых и распределенных приложений. OTP была разработана шведской компанией Ericsson и использовалась внутри Ericsson для разработки отказоустойчивых систем[¹].

OTP содержит ряд мощных инструментов, таких как, полноценный web сервер, FTP сервер, CORBA ORB и других, написанных на Erlang. Еще OTP содержит высокотехнологичные инструменты для создания приложений в сфере телекоммуникаций, с реализацией протоколов H.248, SNMP, и кросс-компилятор ASN.1-to-Erlang. Но я не буду говорить об этом; вы сможете найти информацию по этой теме, посетив сайты, ссылки на которые даны в разделе С.1 *Онлайн документации*, на странице ____.

Если вы хотите разработать свою программу, используя OTP, тогда основные

принципы в поведении OTP будут для вас очень привлекательны. Это поведение объединяет общие поведенческие модели – думайте об этом, как об основе которая, по сути, есть параметризованные *вызовы* модулей. Мощь OTP исходит из ее свойств, таких как отказоустойчивость, масштабируемость, динамический изменяемый код и т.д. собственно это и есть поведение OTP. Другими словами, при написании обратных вызовов вам не надо беспокоиться об отказоустойчивости, потому что об этом позаботится сама OTP. Java-программисты могут думать о поведении как о J2EE контейнере.

Проще говоря, поведение решает нефункциональную часть проблемы, а обратные вызовы – функциональную. Прелесть в том, что нефункциональная часть проблемы (например, динамическое изменение кода) всегда одинакова для всех приложений, тогда как функциональная часть (реализация обратных вызовов) различна в каждом отдельном случае.

В этой главе мы увидим одно из поведений, модуль `gen_server`, во всех деталях. Но перед тем как погрузиться во все тонкости работы `gen_server`, сначала мы рассмотрим простой сервер (простейший сервер, который возможно показать) и будем его изменять шаг за шагом, пока не получим полноценный модуль `gen_server`. Таким образом, вы реально сможете понять, как работает `gen_server` и будете готовы к исследованию внутренностей.

Вот план этой главы:

- Написание маленькой клиент-серверной программы на Erlang.
- Постепенная «генерализация» этой программы и добавление новых возможностей.
- Переход к реальному коду.

16.1 Путь к обыкновенному серверу (Generic Server)

Это наиболее важный подраздел в этой книге, прочитайте его один раз, два раза, прочитайте его 100 раз – чтобы убедиться в том, что вы все поняли.

Мы приступаем к написанию четырех маленьких серверов с названиями `server1`, `server2`..., каждый слегка будет отличаться от предыдущего. Нашей целью является полное разделение нефункциональной и функциональной частей решаемой задачи. Последнее предложение сейчас скорее всего ничего для вас не значит, но не беспокойтесь – скоро об всем узнаете. Итак, глубоко вдохните...

Сервер №1: Простой сервер

Первая попытка. Это маленький сервер, который мы реализуем, написав модуль обратных вызовов.

[Скачать server1.erl](#)

```
-module(server1).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.

loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            {Response, State1} = Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, State1)
    end.
```

Это небольшое количество кода является основой для сервера. Давайте напишем *обратные вызовы для сервера №1*. Вот код модуля обратных вызовов:

[Скачать name_server.erl](#)

```
-module(name_server).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server1, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name)      -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict)     -> {dict:find(Name, Dict), Dict}.
```

Этот код фактически выполняет две задачи. Он выступает в роли модуля обратных

вызовов, вызываемых из серверного кода, и иногда содержит интерфейсные конструкции, которые будут вызываться на стороне клиента. Обычно, по соглашениям OTP, эти функции объединяются в один модуль.

Чтобы увидеть как это работает, сделайте следующее:

```
1> server1:start(name_server, name_server).
true
2> name_server:add(joe, "at home").
ok
3> name_server:whereis(joe).
{ok, "at home"}
```

Сейчас **прервемся и подумаем**. Обратный вызов не имеет кода для параллелизации, не порождает процессы, не отправляет и не принимает сообщения, ничего не регистрирует. Это просто последовательный код и **ничего более**. Что же это значит?

А это означает то, что мы сможем написать клиент-серверное приложение без понимания того, что лежит в основе модели параллельных процессов.

Это **основной шаблон** для всех серверов. Однажды вы поймете основные *структуры*, это просто как «цигарка».

Сервер №2: Сервер с транзакциями

В этом примере сервер прервёт клиента, если результатом запроса будет ошибка:

[Скачать server2.erl](#)

```
-module(server2).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, Request} ->
```

```

try Mod:handle(Request, OldState) of
    {Response, NewState} ->
        From ! {Name, ok, Response},
        loop(Name, Mod, NewState)
    catch
        _:Why ->
            log_the_error(Name, Request, Why),
            %% send a message to cause the client to crash
            From ! {Name, crash},
            %% loop with the *original* state
            loop(Name, Mod, OldState)
    end
end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

Если возникает исключение в обработчике, то единственное что дает нам «транзакционную семантику» в этом сервере – это цикл с *оригинальным значением State*. Если обработчик завершится успешно, то тогда цикл со значением *NewState*, предоставляется обработчику.

Зачем хранить оригинальное состояние? Обработчик выполняется с ошибкой тогда, когда клиент отправляет неверное сообщение, в ответ клиент получает сообщение об аварии. Клиент не может работать, потому что запрос, отправленный на сервер, привел к сбою в обработчике. Зато другие клиенты желающие использовать этот сервер не пострадают. Более того, состояние сервера не изменится, если в обработчике возникнет ошибка.

Замечу, что модуль обратных вызовов для этого сервера *точно такой же как и для сервера №1. Изменяя сервер и оставляя неизменным модуль обратных вызовов, мы может менять нефункциональную часть поведения модуля обратных вызовов.*

Примечание: Последнее высказывание не является чистой правдой. Мы все-таки сделали небольшие изменения в модуле обратных вызовов, когда мы перешли от **сервера №1** к **серверу №2** мы все же изменили имя сервера в директиве `-import` с `server1` на `server2`. Других изменений не было.

Сервер №3: Сервер с горячей заменой кода

Сейчас мы добавим в наш сервер механизм горячей замены кода:

[Скачать server3.erl](#)

```

-module(server3).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name,
        spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallBackMod}} ->
            From ! {Name, ack},
            loop(Name, NewCallBackMod, OldState);
        {From, Request} ->
            {Response, NewState} = Mod:handle(Request, OldState),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.

```

Как же это работает?

Если мы отправляем серверу сообщение о замене кода (swap code), значит мы хотим заменить работающий модуль обратных вызовов на новый модуль, имя которого передается в сообщении. Продемонстрировать это можно запустив `server3` с модулем обратных вызовов и динамический подменить модуль на новый. Мы не сможем использовать `name_server` в качестве модуля обратных вызовов, поскольку это имя сервера и оно жестко задано, так как компилируется внутрь модуля сервера. В итоге нам необходимо сделать копию старого модуля и назвать его `name_server1`, где мы изменим имя сервера:

[Скачать `name_server1.erl`](#)

```

-module(name_server1).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server3, [rpc/2]).


%% client routines

```

```

add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.

```

Сначала мы запустим `server3` с модулем обратных вызовов `name_server1`:

```

1> server3:start(name_server, name_server1).
true
2> name_server:add(joe, "at home").
ok
3> name_server:add(helen, "at work").
ok

```

Теперь, я полагаю, мы захотим найти все имена которые обслуживает наш сервер имен. Но в нашем API нет функции для выполнения такой задачи – модуль `name_server` имеет лишь функции для добавления и поиска имен.

Быстро запускаем наш текстовый редактор и пишем наш новый модуль обратных вызовов:

[Скачать `new_name_server.erl`](#)

```

-module(new_name_server).
-export([init/0, add/2, all_names/0, delete/1, whereis/1, handle/2]).
-import(server3, [rpc/2]).


%% interface
all_names()      -> rpc(name_server, allNames).
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
delete(Name)     -> rpc(name_server, {delete, Name}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle(allNames, Dict)           -> {dict:fetch_keys(Dict), Dict};
handle({delete, Name}, Dict)     -> {ok, dict:erase(Name, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.

```

Сейчас мы скомпилируем этот код и скажем серверу заменить работающий модуль обратных вызовов новым:

```
4> c(new_name_server).  
{ok,new_name_server}  
5> server3:swap_code(name_server, new_name_server).  
Ack
```

И можем запустить новые функции этого сервера:

```
6> new_name_server:all_names().  
[joe,helen]
```

Здесь мы *заменили модуль обратных вызовов «на лету»* - это и есть динамическая замена кода в действии, вы все видели сами и никакой черной магии.

Сейчас прервемся и снова подумаем. Последние две задачи, которые мы с вами решили, в целом, считаются сложными, но на самом деле это очень сложные задачи. Серверы с механизмом транзакций сложны в написании; серверы с динамической заменой кода еще более сложны в написании.

Эта технология чрезвычайно мощна. Обычно мы думаем о серверах как о программах, чье состояние меняется при отправке им сообщений. Код в серверах фиксированный при первом их запуске, и если мы хотим изменить поведение сервера, то нам необходимо остановить сервер, изменить его код и снова его запустить. В этом примере, код сервера может быть изменен также легко, как можно изменить состояние у сервера[^2].

Сервер №4: Транзакции и горячая замена кода

В предыдущих двух серверах семантика горячей замены и семантика транзакций были разделены. Давайте объединим обе возможности в одном сервере. Итак, держите ваши шляпы...

[Скачать server4.erl](#)

```
-module(server4).  
-export([start/2, rpc/2, swap_code/2]).  
  
start(Name, Mod) ->  
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).  
  
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).
```

```

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallbackMod}} ->
            From ! {Name, ok, ack},
            loop(Name, NewCallbackMod, OldState);
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _: Why ->
                    log_the_error(Name, Request, Why),
                    From ! {Name, crash},
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

Этот сервер предоставляет обе возможности, и горячую замену кода и транзакции.
Замечательно.

Сервер №5: Еще больше кайфа

Теперь, получив знания о динамической замене кода, мы можем кайфнуть по полной программе. Сейчас мы рассмотрим сервер, который ничего не делает, пока мы ему не скажем, изменить поведение:

[Скачать server5.erl](#)

```

-module(server5).
-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).
```

```

wait() ->
    receive
        {become, F} -> F()
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.

```

Если мы запустим это сервер и отправим ему сообщение `{become, F}`, то он превратится в F сервер, исполнив `F()`. Запустим сервер:

```

1> Pid = server5:start().
<0.57.0>

```

Наш сервер ничего не делает, он просто ждет сообщение `become`.

Теперь давайте создадим функциональность сервера. Ничего сложного придумывать не будем, просто посчитаем факториал:

[Скачать my_fac_server.erl](#)

```

-module(my_fac_server).
-export([loop/0]).

loop() ->
    receive
        {From, {fac, N}} ->
            From ! {self(), fac(N)},
            loop();
        {become, Something} ->
            Something()
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).

```

Эрланг в PlanetLab

Несколько лет назад, когда мои исследования только начинались, я работал в PlanetLab. Я имел доступ к сети PlanetLab(*) и установил «пустые» Эрланг серверы на все компьютеры (около 450-ти машин). Я не знал что я буду делать с этими

машинами, просто установил серверную инфраструктуру для использования в каких-нибудь целях в будущем.

Так как я запустил серверы, я мог легко сказать, пустым серверам превратиться в серверы, выполняющие реальную работу.

Обычная практика (для начала) – это запустить web-серверы, и установить плагины на web-серверы. Мой подход – отступить на один шаг назад и установить пустые серверы. Потом уже устанавливать плагины web-серверов для превращения пустых серверов в web-серверы. Ведь когда web-сервер станет не нужен, мы можем заставить серверы выполнять что-нибудь еще.

(*) Сеть, широко используемая учеными для тестирования новых сетевых сервисов или модификации уже существующих <http://www.planet-lab.org/>.

Скомпилируйте этот код, теперь вы сможете сказать процессу <0.57.0>, превратиться в факториал-сервер:

```
2> c(my_fac_server).  
{ok,my_fac_server}  
3> Pid ! {become, fun my_fac_server:loop/0}.  
{become,\#Fun<my_fac_server.loop.0>}
```

Теперь, когда наш сервер стал факториал-сервером, мы сделаем вызов:

```
4> server5:rpc(Pid, {fac,30}).  
2652528598121910586363084800000000
```

Наш процесс будет факториал-сервером до тех пор, пока мы не скажем ему стать кем-нибудь другим, отправив ему сообщение {become, Something}.

Как вы увидели в предыдущем примере, мы может иметь широкий диапазон различных типов серверов, с различной семантикой и совершенно удивительными свойствами. Эта технология почти такая же мощная. Используя весь ее потенциал, можно создавать очень маленькие программы удивительной мощности и красоте. Когда мы делали проекты промышленных масштабов с десятками и сотнями программистов, мы не хотели делать некоторые вещи слишком динамичными. Мы хотели добиться баланса между обобщенностью и мощностью и получали нечто подходящее для коммерческих продуктов. Получали код, который меняется от версии к версии, и который превосходно работает, но очень сложен в отладке, если вдруг что-то пойдет не так. Если мы делали много динамических изменений в нашем коде, и это переставало работать, найти причину было очень нелегко.

Примеры серверов в этом разделе не совсем корректны. Они писались, чтобы

показать идеи, но они содержат одну или две чрезвычайно маленьких и тонких ошибки. Я не буде прямо сейчас рассказывать о них, я дам некоторые комментарии по этому поводу в конце главы.

Эрланговый модуль `gen_server` – это что-то вроде логического завершения последовательности достаточно простых серверов (точно таких же которые мы писали на протяжении всей главы).

Он используется в промышленных продуктах, начиная с 1998 года. Сотни серверов могут быть частью одного продукта. Эти серверы будут написаны программистами с использованием обычного последовательного кода. Все ошибки обрабатываются, и все нефункциональное поведение учтено в типовой части сервера.

Итак, сейчас мы совершим огромный прыжок и рассмотрим `gen_server`.

16.2 Начнем с `gen_server`

Я собираюсь окунуть вас в самую глубь проблемы. Вот простой план написания модуля обратных вызовов для `gen_server`, состоящий из трех пунктов:

1. Выбрать имя для модуля обратных вызовов.
2. Написать интерфейсные функции.
3. Написать шесть обязательных функций для модуля обратных вызовов.

На самом деле это очень просто. Не думайте – просто следуйте плану!

Шаг 1: Выбрать имя для модуля обратных вызовов

Мы будем делать очень простую платежную систему. Поэтому назовем модуль `my_bank` [^3].

Шаг 2: Написать интерфейсные конструкции

Мы определим пять интерфейсных конструкций, все они будут в модуле `my_bank`:

`start()` – Открыть банк.

`stop()` – Закрыть банк.

`new_account(Who)` – Создать новый аккаунт.

`deposit(Who, Amount)` – Положить деньги в банк.

`withdraw(Who, Amount)` – Взять деньги, если есть на счету.

Каждая конструкция это ровно одна конструкция для вызова `gen_server`, как показано ниже:

[Скачать my_bank.erl](#)

```
start() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).  
stop() -> gen_server:call(?MODULE, stop).  
  
new_account(Who) -> gen_server:call(?MODULE, {new, Who}).  
deposit(Who, Amount) -> gen_server:call(?MODULE, {add, Who, Amount}).  
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).
```

`gen_server:start_link({local, Name}, Mod, ...)` запускает локальный сервер[^4]. Макрос `?MODULE` содержит имя модуля `my_bank`. `Mod` – это имя модуля обратных вызовов. Остальные аргументы `gen_server:start` мы пока не будем рассматривать.

`gen_server:call(?MODULE, Term)` используется для вызова удаленных процедур сервера.

Шаг 3: Написать конструкции модуля обратных вызовов

Наш модуль обратных вызовов должен экспортовать шесть функций: `init/1`, `handle_call/3`, `handle_cast/2`, `handle_info/2`, `terminate/2`, и `code_change/3`.

Чтобы облегчить жизнь, мы можем использовать один из шаблонов для создания `gen_server`. Вот пример:

[Скачать gen_server_template.mini](#)

```
-module().  
%% gen_server_mini_template  
  
-behaviour(gen_server).  
-export([start_link/0]).  
%% gen_server callbacks  
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,  
        terminate/2, code_change/3]).  
  
start_link() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).  
  
init([]) -> {ok, State}.  
  
handle_call(_Request, _From, State) -> {reply, Reply, State}.  
handle_cast(_Msg, State) -> {noreply, State}.  
handle_info(_Info, State) -> {noreply, State}.
```

```
terminate(_Reason, _State) -> ok.  
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

Этот пример содержит простой скелет, который нужно заполнить, чтобы получить сервер. Ключевое слово `-behaviour` используется компилятором, чтобы знать какие предупреждения и сообщения об ошибках генерировать.

Примечание: Если вы используете emacs, то вы сможете вставить шаблон несколькими командами. Если ваш редактор переключен в режим эрланга, то выберите в меню Erlang -> Skeletons для создания шаблона `gen_server`. Если у вас нет emacs, то не паникуйте. Я включил текст шаблона в конец главы.

Итак, шаблон вставлен, и мы просто отредактируем его куски. Мы имеем все аргументы в интерфейсных конструкциях, согласно аргументам шаблона.

Наиболее важная часть для нас это функция `handle_call/3`. Мы реализуем код трех запросов в нашем интерфейсе. Пока заполним многоточиями некоторые места, как показано ниже:

```
handle_call({new, Who}, From, State} ->  
    Reply = ...  
    State1 = ...  
    {reply, Reply, State1};  
  
handle_call({add, Who, Amount}, From, State} ->  
    Reply = ...  
    State1 = ...  
    {reply, Reply, State1};  
  
handle_call({remove, Who, Amount}, From, State} ->  
    Reply = ...  
    State1 = ...  
    {reply, Reply, State1};
```

Значение `Reply` отправляется обратно клиенту, как результат вызова удаленной процедуры.

`State` это просто переменная, представляющая глобальное состояние сервера, оно было передано серверу. В нашем банковском сервере состояние не меняется; это просто индекс ETS таблицы и он постоянный (хотя содержимое таблицы меняется).

После редактирования кусков кода в шаблоне, мы получили следующий код:

[Скачать my_bank.erl](#)

```

init([]) -> {ok, ets:new(?MODULE, [])}.

handle_call({new, Who}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
        [] -> ets:insert(Tab, {Who, 0}),
                {welcome, Who};
        [_] -> {Who, you_already_are_a_customer}
    end,
    {reply, Reply, Tab};

handle_call({add, Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
        [] -> not_a_customer;
        [{Who, Balance}] ->
            NewBalance = Balance + X,
            ets:insert(Tab, {Who, NewBalance}),
            {thanks, Who, your_balance_is, NewBalance}
    end,
    {reply, Reply, Tab};

handle_call({remove, Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
        [] -> not_a_customer;
        [{Who, Balance}] when X <= Balance ->
            NewBalance = Balance - X,
            ets:insert(Tab, {Who, NewBalance}),
            {thanks, Who, your_balance_is, NewBalance};
        [{Who, Balance}] ->
            {sorry, Who, you_only_have, Balance, in_the_bank}
    end,
    {reply, Reply, Tab};

handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.

handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.

```

Запускаем наш сервер, вызывая `gen_server:start_link(Name, CallBackMod, StartArgs, Opts)`; эта конструкция вызовет в модуле обратных вызовов `Mod:init(StartArgs)`, и должны нам вернуть `{ok, State}`. Значение `State` передается как третий аргумент в `handle_call`.

Отмечу как мы остановим сервер. `handle_call(Stop, From, Tab)` вернет `{stop, normal, stopped, Tab}` при остановке сервера. Второй аргумент (`normal`) используется как первый аргумент в конструкции `my_bank:terminate/2`. Третий

аргумент (`stopped`) становится возвращаемым значением `my_bank:stop()`.

Теперь все готово. Давайте посетим наш банк:

```
1> my_bank:start().
{ok,<0.33.0>}
2> my_bank:deposit("joe", 10).
not_a_customer
3> my_bank:new_account("joe").
{welcome,"joe"}
4> my_bank:deposit("joe", 10).
{thanks,"joe",your_balance_is,10}
5> my_bank:deposit("joe", 30).
{thanks,"joe",your_balance_is,40}
6> my_bank:withdraw("joe", 15).
{thanks,"joe",your_balance_is,25}
7> my_bank:withdraw("joe", 45).
{sorry,"joe",you_only_have,25,in_the_bank}
```

16.3 Структура обратных вызовов gen_server

Теперь вооружившись идеями, можем приступить к более детальному рассмотрению структуры обратных вызовов `gen_server`.

Что же происходит, когда мы запускаем сервер?

Вызов `gen_server:start_link(Name, Mod, InitArgs, Opts)` запускает все. Создается сервер `Name`. Запускается модуль обратных вызовов `Mod`. `Opts` управляют поведением типичного сервера. Здесь может быть протоколирование сообщений, функции отладки, и многое чего еще. Типичный сервер запускается вызовом `Mod:init(InitArgs)`.

Ниже приведен шаблон для `init`:

```
%%-----%
%% Function: init(Args) -> {ok, State} |
%% {ok, State, Timeout} |
%% ignore |
%% {stop, Reason}
%% Description: Initiates the server
%%-----
init([]) ->
    {ok, #state{}}.
```

При нормальном положении дел, мы просто вернем `{ok, State}`. Значение других

аргументов вы можете найти в руководстве по `gen_server`.

Если возвращается значение `{ok, State}`, значит, сервер успешно запущен и его начальное состояние `State`.

Что же происходит, когда мы обращаемся к серверу?

Для обращения к серверу клиентская программа вызывает `gen_server:call(Name, Request)`. В результате будет вызвана функция `handle_call/3` из модуля обратных вызовов.

`handle_call/3` имеет следующий шаблон:

```
%-----  
% Function:  
% handle_call(Request, From, State) -> {reply, Reply, State} |  
% {noreply, Reply, State, Timeout} |  
% {noreply, State} |  
% {noreply, State, Timeout} |  
% {stop, Reason, Reply, State} |  
% {stop, Reason, State}  
% Description: Handling call messages  
%-----  
handle_call(_Request, _From, State) ->  
    Reply = ok,  
    {reply, Reply, State}.
```

`Request` (второй аргумент `gen_server:call/2`) станет первым аргументом `handle_call/3`. `From` – это PID, запрашивающего клиентского процесса, а `State` – это текущее состояние клиента.

Если все хорошо, мы возвращаем `{reply, Reply, NewState}`. Когда это происходит `Reply` уходит обратно к клиенту, где превращается в возвращаемое значение `gen_server:call`. `NewState` – это следующее состояние сервера.

Другие возвращаемые значения `{noreply, ...}` и `{stop, ...}` используются достаточно редко, `noreply` заставляет сервер продолжить работу, но клиент будет ожидать ответа, так как сервер означен отвечать всем клиентам. Вызов `stop` с соответствующими аргументами остановит сервер.

Вызовы и Образы

Мы увидели взаимодействие между `gen_server:call` и `handle_call`. Это то, что используется для реализации вызова удаленных процедур (*remote procedure call*). `gen_server:cast(Name, Name)` реализация образа, который просто вызывается, не

возвращая значений (на самом деле просто сообщение, но обычно это вызов образа из удаленной процедуры).

Соответствующий обратный вызов `handle_cast` показан в шаблоне ниже:

```
%%-----  
% Function: handle_cast(Msg, State) -> {noreply, NewState} |  
% {noreply, NewState, Timeout} |  
% {stop, Reason, NewState}  
% Description: Handling cast messages  
%%-----  
handle_cast(_Msg, State) ->  
    {noreply, NewState}.
```

Обработчик обычно возвращает `{noreply, NewState}`, который меняет состояние сервера или `{stop, ...}`, который останавливает сервер.

Спонтанные сообщения

Функция обратного вызова `handle_info(info, State)` используется для обработки спонтанных сообщений получаемых сервером. Так что же такое спонтанные сообщения? Если сервер связан с другими процессами и перехватывает их, он может внезапно принять неожиданное сообщение `{'EXIT', Pid, What}`. Либо, любой процесс в системе, который знает PID сервера, может просто отправить ему сообщение. Любые такие сообщения будут приняты сервером как значение переменной `Info`.

Шаблон для `handle_info` выглядит так:

```
%%-----  
% Function: handle_info(Info, State) -> {noreply, State} |  
% {noreply, State, Timeout} |  
% {stop, Reason, State}  
% Description: Handling all non-call/cast messages  
%%-----  
handle_info(_Info, State) ->  
    {noreply, State}.
```

Возвращаемое значение такое же как у `handle_cast`.

Прощай, малышка

Сервер может прервать свою работу по многим причинам. Один из `handle_Something` вызовов может вернуть `{stop, Reason, NewState}`, либо сервер может рухнуть при сообщении `{'Exit', reason}`. При любом раскладе будет вызвана функция

```
terminate(Reason, NewState).
```

Вот ее шаблон:

```
%%-----  
%% Function: terminate(Reason, State) -> void()  
%% Description: This function is called by a gen_server when it is  
%% about to terminate. It should be the opposite of Module:init/1 and  
%% do any necessary  
%% cleaning up. When it returns, the gen_server terminates with Reason.  
%% The return value is ignored.  
%%-----  
terminate(_Reason, State) ->  
ok.
```

Эта функция не возвращает новое состояние, потому что вся работа уже прервана. И что же можно сделать в этом состоянии (`State`)? Оказывается многое. Мы можем сохранять его на диск, Отправить в сообщении другим процессам или отказаться от него, если это необходимо приложению. Если вы хотите чтобы ваш сервер был когда-нибудь перезапущен, вам необходимо написать функцию «Я еще вернусь», которую вызовет `terminate/2`.

Замена кода

Вы можете динамически изменять состояние своего сервера, пока он запущен. Вызов этой функции обратного вызова производит подсистема "управления релизами" когда система выполняет обновление программного кода.

Эта часть детально описана в разделе "Управление релизами" в документации о принципах дизайна OTP[^5].

```
%%-----  
%% Func: code_change(OldVsn, State, Extra) -> {ok, NewState} %%  
%% Description: Convert process state when code is changed  
%%-----  
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

16.4 Код и Шаблоны

Этот код сделан в emacs:

gen_server template

[Скачать gen_server_template.full](#)

```
%%%-----  
%%% File      : gen_server_template.full  
%%% Author    : my name <yourname@localhost.localdomain>  
%%% Description :  
%%%  
%%% Created : 2 Mar 2007 by my name <yourname@localhost.localdomain>  
%%%-----  
-module().  
  
-behaviour(gen_server).  
  
%% API  
-export([start_link/0]).  
  
%% gen_server callbacks  
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,  
        terminate/2, code_change/3]).  
  
-record(state, {}).  
  
%%=====  
%% API  
%%=====  
%%-----  
%% Function: start_link() -> {ok,Pid} | ignore | {error,Error}  
%% Description: Starts the server  
%%-----  
start_link() ->  
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).  
  
%%=====  
%% gen_server callbacks  
%%=====  
  
%%-----  
%% Function: init(Args) -> {ok, State} |  
%%                      {ok, State, Timeout} |  
%%                      ignore          |  
%%                      {stop, Reason}  
%% Description: Initiates the server  
%%-----  
init([]) ->  
    {ok, #state{}}.
```

```

%-----
%% Function: %% handle_call(Request, From, State) -> {reply, Reply, State}
%%                                         {reply, Reply, State, Timeout} |
%%                                         {noreply, State} |
%%                                         {noreply, State, Timeout} |
%%                                         {stop, Reason, Reply, State} |
%%                                         {stop, Reason, State}
%% Description: Handling call messages
%-----

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

%-----
%% Function: handle_cast(Msg, State) -> {noreply, State} |
%%                                         {noreply, State, Timeout} |
%%                                         {stop, Reason, State}
%% Description: Handling cast messages
%-----

handle_cast(_Msg, State) ->
    {noreply, State}.

%-----
%% Function: handle_info(Info, State) -> {noreply, State} |
%%                                         {noreply, State, Timeout} |
%%                                         {stop, Reason, State}
%% Description: Handling all non call/cast messages
%-----

handle_info(_Info, State) ->
    {noreply, State}.

%-----
%% Function: terminate(Reason, State) -> void()
%% Description: This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any necessary
%% cleaning up. When it returns, the gen_server terminates with Reason.
%% The return value is ignored.
%-----

terminate(_Reason, _State) ->
    ok.

%-----
%% Func: code_change(OldVsn, State, Extra) -> {ok, NewState}
%% Description: Convert process state when code is changed

```

```
%%-----  
code_change(_OldVsn, State, _Extra) ->  
{ok, State}.  
  
%%-----  
%%% Internal functions  
%%-----
```

my_bank

Скачать [my_bank.erl](#)

```
-module(my_bank).  
  
-behaviour(gen_server).  
-export([start/0]).  
%% gen_server callbacks  
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,  
        terminate/2, code_change/3]).  
-compile(export_all).  
  
start() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).  
stop() -> gen_server:call(?MODULE, stop).  
  
new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).  
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).  
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).  
  
init([]) -> {ok, ets:new(?MODULE, [])}.  
  
handle_call({new, Who}, _From, Tab) ->  
    Reply = case ets:lookup(Tab, Who) of  
        []  -> ets:insert(Tab, {Who, 0}),  
                {welcome, Who};  
        [_] -> {Who, you_already_are_a_customer}  
    end,  
    {reply, Reply, Tab};  
handle_call({add, Who, X}, _From, Tab) ->  
    Reply = case ets:lookup(Tab, Who) of  
        []  -> not_a_customer;  
        [{Who, Balance}] ->  
            NewBalance = Balance + X,  
            ets:insert(Tab, {Who, NewBalance}),
```

```

        {thanks, Who, your_balance_is, NewBalance}
    end,
{reply, Reply, Tab};
handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
        [] -> not_a_customer;
        [{Who,Balance}] when X =< Balance ->
            NewBalance = Balance - X,
            ets:insert(Tab, {Who, NewBalance}),
            {thanks, Who, your_balance_is, NewBalance};
        [{Who,Balance}] ->
            {sorry,Who,you_only_have,Balance,in_the_bank}
    end,
{reply, Reply, Tab};
handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.

handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.

```

16.5 Копаем глубже

Мы увидели, что `gen_server` достаточно прост. Мы не рассмотрели некоторые интерфейсные функции `gen_server`-а, и не поговорили обо всех аргументах интерфейсных функций. Если вы поняли основные идеи, то с деталями разберетесь, обратившись к документации по `gen_server`.

В этой главе мы увидели только простейшие возможные пути использования `gen_server`, но этого должно быть достаточно для решения большинства задач. Более сложные приложения позволяют `gen_server`-у отвечать со значением `noreply` и делегировать ответ другому процессу. Информацию об этом вы можете прочитать в главе "Принципы дизайна"⁶ и в руководстве по модулям `sys` и `proc_lib`.

Глава 17. Mnesia: СУБД для Эрланга (и на Эрланге)

Предположим, вы захотели создать многопользовательскую игру, сделать новый веб-сайт, или разработать систему онлайн-платежей. Вероятно, вам в этом случае

понадобится какая-либо СУБД.

Среди тысячи-другой файлов, появившихся на диске после установки Эрланга, присутствует полнофункциональная СУБД под названием Mnesia. Она исключительно быстра, и она может хранить структуры, состоящие из любых доступных в Эрланге типов данных.

Кроме того, Mnesia поддаётся детальной настройке и конфигурированию.

Определённые таблицы могут храниться в RAM (для обеспечения высокой скорости доступа), другие - сохраняются на диск (выживая при падении узла), а ещё таблицы могут реплицироваться на другие компьютеры в сети, что даёт возможность строить устойчивые к сбоям системы.

Давайте познакомимся со всем этим поближе.

17.1 Запросы к базе данных

Начнём, пожалуй, с изучения запросов к БД Mnesia. При ближайшем рассмотрении оказывается, что запросы Mnesia одновременно похожи и на SQL, и на обработчики запросов (`list comprehensions`), так что нам не придётся изучать так уж много нового.)

Во всех примерах я буду подразумевать, что вы создали базу данных с двумя таблицами с названиями `shop` и `cost`. Эти таблицы содержат следующие данные.

Таблица `shop`:

Item	Quantity	Cost
apple	20	2.3
orange	100	3.8
pear	200	3.6
banana	420	4.5
potato	2456	1.2

Таблица `cost`:

Name	Price
apple	1.5
orange	2.4
pear	2.2

banana	1.5
otato	0.6

Чтобы представить эти таблицы в подходящем для Mnesia виде, мы должны создать определения записей, которые бы описывали колонки в таблицах. Определения эти таковы:

[Скачать test_mnesia.erl](#)

```
-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
```

Теперь немного чёрной магии. Я собираюсь показать, как работают запросы, и хочу, чтобы вы могли повторить это все у себя дома. Но для этого сначала кто-то должен создать и заполнить для вас базу данных. Так что пока поверьте мне на слово - в файле [test_mnesia.erl](#) я подготовил код, который производит инициализацию, и вы можете просто запустить его из оболочки `erl`.

```
1> c(test_mnesia).
{ok,test_mnesia}
2> test_mnesia:do_this_once().
=INFO REPORT==== 29-Mar-2007::20:33:12 ===
    application: mnesia
    exited: stopped
    type: temporary
stopped
```

Теперь мы можем приступить к нашим примерам.

Выборка всех данных из таблицы

Вот так выглядит код для выборки всех данных из таблицы `shop` (для тех, кто знаком с SQL, каждый фрагмент кода начинается с комментария, где показан соответствующий задаче оператор SQL).

[Скачать test_mnesia.erl](#)

```
%> %% SQL equivalent
%> %% SELECT * FROM shop;
demo(select_shop) ->
    do qlc:q([X || X <- mnesia:table(shop)]));
```

Самая суть примера заключена в вызове функции `qlc:q`, которая компилирует запрос (её параметр) во внутреннее представление, которое уже используется для

непосредственного выполнения запроса.

Мы передаем полученный запрос в функцию с названием `do()`, которая определена в модуле `test_mnesia` (ближе к концу модуля). Она отвечает за выполнение запроса и возврат результата.

Для того, чтобы её можно было легко вызывать из оболочки `erl`, мы заключили этот код в функцию `demo(select_shop)` (Полный листинг модуля `mnesia_test` приведён в конце главы).

Мы можем вызвать её таким образом:

[Скачать test_mnesia.erl](#)

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

Примечание: выбранные из таблицы строки могут оказаться в другом порядке.

Строка, которая задает запрос, в этом примере выглядит так:

```
qlc:q([X || X <- mnesia:table(shop)])
```

Это очень похоже на обработчики списков (`list comprehensions`) (см. раздел 3.6, *Обработчики списков*, на странице [__](#)). На самом деле, `qlc` расшифровывается как *"query list comprehensions"*. Это один из модулей, которые служат для доступа к данным в БД Mnesia.

Выражение `[X || X <- mnesia:table(shop)]` означает “список `X`, таких, что `X` берется из таблицы Mnesia с названием `shop`”. Значения `X` - это Эрланг-записи типа `shop`.

Важно: аргумент функции `qlc:q/1` должен быть литералом обработчика списков (`list comprehension`, а не чем-либо, что вычисляется в такое выражение. Таким образом, например, вот такой код не будет эквивалентным тому, что написано в примере:

```
Var = [X || X <- mnesia:table(shop)],  
qlc:q(Var)
```

Выборка определенных колонок из таблицы (проекция)

Следующий запрос выбирает колонки `item` и `quantity` из таблицы `shop`.

Скачать [test_mnesia.erl](#)

```
%% SQL equivalent  
%% SELECT item, quantity FROM shop;  
  
demo(select_some) ->  
    do qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)]);  
  
4> test_mnesia:demo(select_some).  
[{orange,100},{pear,200},{banana,420},{potato,2456},{apple,20}]
```

В предыдущем примере значения `X` были записями типа `shop`. Если вы вспомните синтаксис выражений с записями из раздела 3.9, Записи, на странице ___, вы увидите, что `X#shop.item` соответствует полю `item` записи типа `shop`. Таким образом, кортеж `{X#shop.item, X#shop.quantity}` состоит из двух полей записи `X` - `item` и `quantity`.

Выборка из таблицы данных, удовлетворяющих условию

Следующий запрос выберет все наименования товаров из таблицы `shop`, где количество товара на складе меньше, чем 250. Например, мы могли бы использовать этот вопрос, чтобы решить, какие товары нам надо дозаказать.

Скачать [test_mnesia.erl](#)

```
%% SQL equivalent  
%% SELECT shop.item FROM shop  
%% WHERE shop.quantity < 250;  
demo(reorder) ->  
    do qlc:q([X#shop.item || X <- mnesia:table(shop),  
              X#shop.quantity < 250  
            ]);  
5> test_mnesia:demo(reorder).  
[orange,pear,apple]
```

Обратите внимание, как наше условие естественно вписалось в обработчик списков.

Выборка данных из двух таблиц (Join)

Теперь предположим, что мы хотим дозаказать товар только в том случае, если на складе его осталось менее 250 штук, а цена его - меньше чем 2.0 за штуку. Чтобы это сделать, нам придётся обратиться сразу к двум таблицам. Вот так выглядит нужный запрос:

Скачать [test_mnesia.erl](#)

```
%% SQL equivalent
%% SELECT shop.item, shop.quantity, cost.name, cost.price
%% FROM shop, cost
%% WHERE shop.item = cost.name
%% AND cost.price < 2
%% AND shop.quantity < 250

demo(join) ->
    do qlc:q([X#shop.item || X <- mnesia:table(shop),
              X#shop.quantity < 250,
              Y <- mnesia:table(cost),
              X#shop.item == Y#cost.name,
              Y#cost.price < 2
            ])).
6> test_mnesia:demo(join).
[apple]
```

Здесь важным фрагментом является соединение между наименованием товара в таблице `shop` и наименованием товара в таблице `cost`:

```
X#shop.item == Y#cost.name
```

17.2 Вставка и удаление данных в/из БД.

Снова предполагаем, что мы создали БД и определили таблицу `shop`. Теперь мы хотим добавить строку в таблицу, или, наоборот, удалить строку из таблицы.

Вставка строки

Мы можем добавить строку в таблицу `shop` следующим образом:

Скачать [test_mnesia.erl](#)

```
add_shop_item(Name, Quantity, Cost) ->
```

```
Row = #shop{item=Name, quantity=Quantity, cost=Cost},  
F = fun() ->  
    mnesia:write(Row)  
end,  
mnesia:transaction(F).
```

Эта функция создаёт запись типа `shop` и вставляет её в таблицу:

```
1> test_mnesia:start().  
ok  
2> test_mnesia:reset_tables().  
{atomic, ok}  
% list the shop table  
3> test_mnesia:demo(select_shop).  
[{shop,orange,100,3.80000},  
 {shop,pear,200,3.60000},  
 {shop,banana,420,4.50000},  
 {shop,potato,2456,1.20000},  
 {shop,apple,20,2.30000}]  
% add a new row  
4> test_mnesia:add_shop_item(orange, 236, 2.8).  
{atomic,ok}  
% list the shop table again so we can see the change  
5> test_mnesia:demo(select_shop).  
[{shop,orange,236,2.80000},  
 {shop,pear,200,3.60000},  
 {shop,banana,420,4.50000},  
 {shop,potato,2456,1.20000},  
 {shop,apple,20,2.30000}]
```

Примечание: первичным ключом в таблице `shop` является первая колонка таблицы, то есть, наименование товара (`item`) в записи `shop`. Таблица создана с типом "set" (см. обсуждение типов `set` и `bag` в разделе 15.2, *Types of Table*, стр. ____). Если вновь созданная запись имеет такое же значение первичного ключа, как у уже существующей в БД строки, она перезапишет эту строку, в противном случае в БД добавится новая строка.

Удаление строки

Чтобы удалить строку, нам нужно знать ID объекта (`OID`) для этой строки. `OID` составляется из имени таблицы и значения первичного ключа:

Скачать [test_mnesia.erl](#)

```
remove_shop_item(Item) ->
```

```

Oid = {shop, Item},
F = fun() ->
    mnesia:delete(Oid)
end,
mnesia:transaction(F).

6> test_mnesia:remove_shop_item(pear).
{atomic,ok}
% list the table -- the pear has gone
7> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
 [{}]
8> mnesia:stop().
ok

```

17.3 Транзакции в Mnesia

Когда ранее мы добавляли или удаляли строки в/из БД, или выполняли запрос к БД, мы писали что-то вроде такого:

```

do_something(...) ->
F = fun() ->
    % ...
    mnesia:write(Row)
    % ... или ...
    mnesia:delete(Oid)
    % ... или ...
    qlc:e(Q)
end,
mnesia:transaction(F)

```

`F` — это анонимная функция без аргументов. Внутри `F` мы вызывали некое сочетание функций `mnesia:write/1`, `mnesia:delete/1` или `qlc:e(Q)` (где `Q` это запрос, предварительно скомпилированный при помощи `qlc:q/1`).

Зачем мы так делали? Что вообще значит "транзакция"? Чтобы пояснить это, предположим, что у нас есть два процесса, которые одновременно пытаются получить доступ к одним и тем же данным. Например, у меня есть 10 долларов на банковском счете. Теперь предположим, что два человека пытаются одновременно взять по 8 долларов с этого счета. В данном случае я бы хотел, чтобы одно из этих снятий

произошло успешно, а другое - не произошло.

Именно эту гарантию предоставляет нам `mnesia:transaction/1`. Либо все чтения и записи данных в БД в пределах одной конкретно взятой транзакции завершаются успешно, либо не завершается успешно ни одна из этих операций. Если ни одна из составляющих транзакцию операций не успешна, мы говорим, что транзакция терпит неудачу. Если транзакция терпит неудачу, никаких изменений в БД не производится.

Стратегия, которой при этом следует Mnesia, является разновидностью пессимистических блокировок. Всякий раз, когда менеджер транзакций Mnesia обращается к таблице, он пытается заблокировать запись или всю таблицу, в зависимости от ситуации. Если он обнаруживает, что блокировка может привести к взаимоблокировке (`deadlock`), он немедленно отменяет транзакцию и откатывает все изменения, которые в пределах этой транзакции были сделаны. Если транзакция с самого начала терпит неудачу по причине того, что другой процесс занял данные, система делает небольшую паузу и повторяет транзакцию. Одним из результатов такого поведения является то, что код внутри анонимной функции транзакции может выполняться большое количество раз.

Именно по этой причине анонимная функция транзакции не должна делать ничего такого, что приводит к побочным эффектам. Например, если бы мы написали что-то такое:

```
F = fun() ->
    ...
    io:format("reading ..."), % don't do this
    ...
end,
mnesia:transaction(F),
```

мы могли бы получить на выводе большую кучу ненужного текста, просто потому что функция была перезапущена много раз.

Примечание 1: функции `mnesia:write/1` и `mnesia:delete/1` должны вызываться только в анонимной функции, которую выполняет `mnesia:transaction/1`.

Примечание 2: Никогда не пишите код, который явно перехватывает исключения от функций доступа Mnesia (`mnesia:write/1`, `mnesia:delete/1`, и т.п.), поскольку механизм транзакций Mnesia сам основан на том, что эти функции выбрасывают исключения при неудаче. Если вы будете перехватывать исключения и обрабатывать их сами, вы сломаете механизм транзакций.

Отмена транзакции

Около нашего магазина есть ферма, где выращивают яблоки. Хозяин фермы любит апельсины, и покупает их у нас по бартеру, на яблоки. "Курс обмена" - два к одному, то есть, чтобы купить ' N ' апельсинов, фермер даёт нам $2*N$ яблок.

Вот так выглядит функция, которая обновляет БД, когда фермер покупает апельсины:

[Скачать test_mnesia.erl](#)

```
farmer(Nwant) ->
% Nwant = Number of oranges the farmer wants to buy
F = fun() ->
    %% find the number of apples
    [Apple] = mnesia:read({shop,apple}),
    Napples = Apple#shop.quantity,
    Apple1 = Apple#shop{quantity = Napples + 2*Nwant},
    %% update the database
    mnesia:write(Apple1),
    %% find the number of oranges
    [Orange] = mnesia:read({shop,orange}),
    NOranges = Orange#shop.quantity,
    if
        NOranges >= Nwant ->
            N1 = NOranges - Nwant,
            Orange1 = Orange#shop{quantity=N1},
            %% update the database
            mnesia:write(Orange1);
        true ->
            %% Oops -- not enough oranges
            mnesia:abort(oranges)
    end
end,
mnesia:transaction(F).
```

Этот код, честно говоря, глуповат, но я так написал специально, чтобы продемонстрировать работу транзакций. Сначала я изменяю количество яблок в БД. Это делается до того, как я проверяю количество доступных апельсинов. Это сделано специально, чтобы показать, что изменения можно "откатить", если транзакция неудачна. В обычной ситуации я бы отложил запись количества яблок, пока не убедился бы в наличии достаточного количества апельсинов. Посмотрим на этот код в действии. Утром фермер приходит и покупает 50 апельсинов.

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
```

```
{atomic, ok}
% List the shop table
3> test_mnesia:demo(select_shop).
[{"shop", "orange", 100, 3.80000},
 {"shop", "pear", 200, 3.60000},
 {"shop", "banana", 420, 4.50000},
 {"shop", "potato", 2456, 1.20000},
 {"shop", "apple", 20, 2.30000}]
% The farmer buys 50 oranges
% paying with 100 apples
4> test_mnesia:farmer(50).
{atomic,ok}
% Print the shop table again
5> test_mnesia:demo(select_shop).
[{"shop", "orange", 50, 3.80000},
 {"shop", "pear", 200, 3.60000},
 {"shop", "banana", 420, 4.50000},
 {"shop", "potato", 2456, 1.20000},
 {"shop", "apple", 120, 2.30000}]
```

После обеда он приходит снова и хочет купить ещё 100 апельсинов (ничего себе, как же он любит апельсины!)

```
6> test_mnesia:farmer(100).
{aborted,oranges}
7> test_mnesia:demo(select_shop).
[{"shop", "orange", 50, 3.80000},
 {"shop", "pear", 200, 3.60000},
 {"shop", "banana", 420, 4.50000},
 {"shop", "potato", 2456, 1.20000},
 {"shop", "apple", 120, 2.30000}]
```

Почему СУБД называется Mnesia?

Изначально её название было "Amnesia". Одному из наших боссов это не понравилось, он сказал: "Нельзя называть её Amnesia - мы не можем выпускать базу данных, которая, судя по названию, забывает всё подряд!" Так что мы убрали букву "A", и название прижилось.

Когда транзакция отменилась (когда мы вызвали `mnesia:abort(Reason)`), те изменения, которые были сделаны при помощи `mnesia:write`, были также отменены. Таким образом, состояние СУБД было восстановлено таким, каким оно было до начала транзакции.

Загрузка тестовых данных

Теперь, когда мы знаем, как работают транзакции, мы можем посмотреть на код, который загружал нам тестовые данные. Функция `test_mnesia:example_tables/0` задает данные, необходимые для инициализации таблиц. Первый элемент кортежа задает имя таблицы, следом идет содержимое строки, в том же порядке, в котором поля были описаны при определении Эрланг-записи.

Скачать [test_mnesia.erl](#)

```
example_tables() ->
    [% The shop table
     {shop, apple, 20, 2.3},
     {shop, orange, 100, 3.8},
     {shop, pear, 200, 3.6},
     {shop, banana, 420, 4.5},
     {shop, potato, 2456, 1.2},
    %% The cost table
     {cost, apple, 1.5},
     {cost, orange, 2.4},
     {cost, pear, 2.2},
     {cost, banana, 1.5},
     {cost, potato, 0.6}
    ].
```

Следующий код вставляет данные в Mnesia из таблиц примера.

```
reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
        foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).
```

Он всего лишь вызывает `mnesia:write` для каждого кортежа из списка, который возвращает `example_tables/1`

Функция `do()`

Функция `do`, которую вызывает `demo/1`, чуть сложнее:

Скачать [test_mnesia.erl](#)

```
do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
```

Val.

Она вызывает `qlc:e(Q)` в рамках транзакции Mnesia. `Q` - это скомпилированный запрос `QLC`, а `qlc:e(Q)` выполняет запрос и возвращает все ответы на запрос в виде списка. Возвращаемое значение `{atomic, Val}` означает, что транзакция завершилась успешно со значением `Val`. `Val` - это значение, возвращаемое анонимной функцией транзакции.

17.4 Сохранение сложных типов данных в Mnesia.

Если вы программируете на C, то как бы вы сохранили структуру С (`struct`) в базе данных SQL? Или, если вы программируете на Java, как бы вы стали сохранять в такой же базе объект? Ответ - с изрядными сложностями. Одной из проблем при использовании привычных СУБД является ограниченность встроенных типов колонок. Вы можете хранить в колонке БД целое число, строку, число с плавающей точкой, и т.п. Но если вы хотите сохранить сложный объект, это реальная проблема.

Mnesia разработана так, чтобы хранить структуры данных Эрланг. На самом деле, вы можете сохранять в таблице Mnesia любую структуру данных, которую можно создать в Эрланг. Чтобы проиллюстрировать это, предположим, что некоторое количество архитекторов хотят сохранять в Mnesia свои чертежи. Для начала мы должны определить запись (`record`), которая будет представлять собой чертёж.

Скачать [test_mnesia.erl](#)

```
-record(design, {id, plan}).
```

Затем мы пишем функцию, которая добавляет несколько чертежей в БД:

Скачать [test_mnesia.erl](#)

```
add_plans() ->
    D1 = #design{id = {joe,1},
                 plan={circle,10}},
    D2 = #design{id = fred,
                 plan={rectangle,10,5}},
    D3 = #design{id = {jane,{house,23}}},
         plan = {house,
                  [{floor,1,
                     [{doors,3},
                      {windows,12},
                      {rooms,5}]}]},
```

```

{floor,2,
 [{doors,2},
  {rooms,4},
  {windows,15}]]]}},
F = fun() ->
    mnesia:write(D1),
    mnesia:write(D2),
    mnesia:write(D3)
end,
mnesia:transaction(F).
```

```

Теперь мы можем добавить несколько чертежей в БД

```

1> test_mnesia:start().
ok
2> test_mnesia:add_plans().
{atomic,ok}

```

Теперь мы имеем несколько чертежей, сохраненных в БД. Мы можем извлекать их оттуда при помощи следующей функции:

```

get_plan(PlanId) ->
 F = fun() -> mnesia:read({design, PlanId}) end,
 mnesia:transaction(F).

3> test_mnesia:get_plan(fred).
{atomic,[{design,fred,{rectangle,10,5}}]}
4> test_mnesia:get_plan({jane, {house,23}}).
{atomic,[{design,{jane,{house,23}}},
 {house,[{floor,1,[{doors,3},
 {windows,12},
 {rooms,5}]}],
 {floor,2,[{doors,2},
 {rooms,4},
 {windows,15}]}]}]}

```

Как вы можете видеть, и ключ, и извлекаемое значение могут быть произвольными термами Эрланг.

Говоря техническим языком, между структурами данных в БД и структурами в Эрланге отсутствует "разность сопротивлений". Это означает, что вставка и удаление сложных структур данных в/из БД работают очень быстро.

## Фрагментированные таблицы

Mnesia поддерживает "фрагментированные таблицы" ( горизонтальноеパーティционирование, если пользоваться привычной терминологией СУБД). Это сделано для поддержки чрезвычайно больших таблиц. Таблицы разделены на фрагменты, которые сохраняются на разных компьютерах. Фрагменты сами по себе являются таблицами Mnesia. Фрагменты могут быть реплицируемы, могут иметь индексы, и тому подобное, как любые другие таблицы.

Обратитесь к документации Mnesia User Guide за дальнейшей информацией.

## 17.5 Типы и расположения таблиц

Мы можем настраивать таблицы Mnesia многими разными способами. Во-первых, таблицы могут храниться в RAM, на диске, и одновременно в RAM+на диске. Во-вторых, таблицы могут располагаться на одной машине или реплицироваться на несколько машин. Когда мы проектируем нашу БД, мы должны думать о том, какого типа данные мы хотим хранить в тех или иных таблицах. Свойства таблицы могут быть такими:

**RAM tables** — Эти таблицы работают очень быстро. Данные в них непостоянны, так как теряются при сбое или перезагрузке компьютера, или когда вы останавливаете СУБД.

**Disk tables** — Хранимые на диске таблицы должны выживать при сбоях (при условии, что диск физически не пострадал).

Когда транзакция записывает данные в таблицу, которая хранится на диске, реально данные, затронутые транзакцией, сначала пишутся в дисковый журнал. Этот журнал постоянно пополняется, и через регулярные промежутки времени его информация консолидируется с другими данными в БД, после чего обработанные записи из журнала стираются. Если система падает, то при перезагрузке дисковый журнал проверяется на актуальность, и все необработанные записи из журнала заносятся в БД прежде чем БД становится доступной для работы. Как только транзакция успешно завершилась, её данные должны быть надлежащим образом записаны в дисковый журнал, и если после этого система упала, а потом перезагружена, изменения, сделанные транзакцией, при этом не потеряются.

Если же система упала во время транзакции, то все изменения, сделанные в пределах этой транзакции, должны быть потеряны.

Прежде чем использовать таблицы в RAM, нужно экспериментально убедиться, что вся таблица убирается в физически доступный объём памяти. Если таблица не влезает в физическую память, система будет активно использовать файл подкачки, а

это обычно убивает производительность.

Поскольку таблицы RAM непостоянны, мы должны спросить себя - будет ли нам плохо, если все данные в нашей таблице RAM вдруг исчезнут? Если ответ "да", то нам нужно реплицировать нашу таблицу на диск, или реплицировать её на другой компьютер (как таблицу RAM, как дисковую таблицу, или RAM+диск).

## Создание таблиц

Для того, чтобы создать таблицу, мы вызываем `mnesia:create_table(Name, Args)`, где `Args` - это список кортежей вида `{Key, Val}`. `create_table` возвращает `{atomic, ok}`, если таблица была успешно создана; если нет - возвращает `{aborted, Reason}`.

Вот некоторые часто используемые аргументы для создания таблиц:

`Name` — Задаёт имя создаваемой таблицы (это должен быть `atom`). По соглашению, это также имя соответствующей Эрланг-записи (`record`) - каждая строка в таблице будет являться экземпляром этой записи.

`{type, Type}` — Задаёт тип таблицы. Тип может быть одним из следующих: `set`, `ordered_set`, или `bag`. Эти типы имеют такое же значение, как описано в разделе 15.2, *Types of Table*, на странице [\\_\\_\\_\\_\\_](#).

`{disc_copies, NodeList}` — `NodeList` задаёт список узлов Эрланг, на которых будут храниться дисковые копии создаваемой таблицы. Когда мы используем эту опцию, система также создаёт RAM-копию таблицы на том узле, на котором было выполнено создание таблицы. В принципе, допустимо иметь реплицируемую таблицу типа `disc_copies` на одном узле, плюс иметь ту же таблицу с другим типом на другом узле. Это делается, если мы хотим добиться следующего:

1. Чтобы операции чтения были очень быстрыми и выполнялись из RAM
2. При этом операции записи выполнялись бы в постоянное хранилище.

`{ram_copies, NodeList}` — `NodeList` задаёт список узлов Эрланг, на которых будут храниться копии в RAM.

`{disc_only_copies, NodeList}` — `NodeList` задаёт список узлов Эрланг, на которых будут храниться копии данных в режиме только-на-диск. Такие таблицы не имеют реплики в RAM, и, соответственно, доступ к ним медленнее.

`{attributes, AtomList}` — `AtomList` задаёт список имен колонок в данной таблице. Заметим, чтобы создать таблицу, содержащую экземпляры записи (`record`) с именем `xxx`, мы можем использовать синтаксис: `{attribute,`

```
record_info(fields, xxx)} (конечно, по-другому, можно явно указать список имен).
```

*Примечание:* у функции `create_table` есть и другие опции, здесь я показал не все. За более точной информацией обратитесь к документации по Mnesia.

## Часто используемые комбинации атрибутов таблиц

Во всех дальнейших примерах мы будем предполагать, что `Attrs` является кортежем вида `{attributes, ...}`.

Приведём здесь некоторые часто используемые опции настройки таблиц, покрывающих большинство типовых случаев:

```
mnesia:create_table(shop, [Attrs])
```

- Таблица хранится в RAM на единственном узле.
- Если узел падает, таблица будет потеряна.
- Самый быстрый из всех методов.
- Таблица должна помещаться в физическую память.

```
mnesia:create_table(shop,[Attrs,{disc_copies,[node()]}])
```

- Таблица располагается в RAM + копия на диске, на единственном узле.
- Если узел падает, таблица будет восстановлена с диска.
- Быстрые операции чтения, запись более медленная.
- Таблица должна помещаться в физическую память.

```
mnesia:create_table(shop, [Attrs,{disc_only_copies,[node()]}])
```

- Копия только-на-диске на единственном узле.
- Большие таблицы не обязаны помещаться полностью в память.
- Доступ медленнее, чем к таблицам с репликой в RAM.

```
mnesia:create_table(shop,[Attrs,{ram_copies,[node(),someOtherNode()]}])
```

- Таблица хранится в RAM на двух разных узлах.
- Если оба узла падают, таблица будет потеряна.
- Таблица должна помещаться в физическую память.
- К таблице можно обращаться с любого из этих двух узлов.

```
mnesia:create_table(shop,[Attrs, {disc_copies, [node(),someOtherNode()]}])
```

- Дисковые копии на двух узлах.
- Таблица может быть восстановлена на любом из этих двух узлов.

- Таблица выживает после двойного сбоя.

## Поведение таблиц при репликации

Когда таблица реплицируется на несколько узлов Эрланг, она синхронизируется, пока это возможно. Если один узел падает, система продолжает работать, но количество реплик уменьшается. Когда упавший узел возвращается в строй, он "догонит" состояние таблицы, обратившись к другим узлам, где хранятся актуальные реплики.

Заметка: Mnesia может оказаться перегруженной, если узлы, на которых она запущена, прекращают работать. Если вы используете ноутбук, который "засыпает" при неактивности, то при "просыпании" Mnesia может оказаться временно перегруженной и выдавать множество сообщений о предупреждениях. На эти предупреждения можно не обращать внимания.

---

## 17.6 Первоначальное создание базы данных

Вот так выглядят команды, создающие базу данных Mnesia. Вам нужно выполнить это лишь один раз.

```
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.nonode@nohost
```

Вызов функции `mnesia:create_schema(NodeList)` инициализирует новую базу данных Mnesia на всех узлах из списка `NodeList` (он должен содержать список реально существующих Эрланг-узлов). В нашем случае мы задаём список узлов, как `[node()]`, то есть, содержащим лишь один, текущий, узел. Mnesia инициализируется и создаёт на диске структуру каталогов с названием `Mnesia.nonode@nohost`, где физически хранит базу данных. Мы можем выйти из оболочки и выполнить команду `ls`, чтобы проверить это.

Если мы повторим это упражнение на запущенном в распределённом режиме узле с именем `joe`, получим такой результат:

```
$ erl -name joe
(joe@doris.myerl.example.com) 1> mnesia:create_schema([node()]).
mnesia:create_schema([node()]).
```

```

ok
2> init:stop().
ok
$ ls
Mnesia.joe@doris.myerl.example.com

```

С другой стороны, мы можем сами указать путь к нужной нам конкретной базе данных при старте Эрланга:

```

$ erl -mnesia dir '"${home}/joe/some/path/to/Mnesia.company"'
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok

```

`"/home/joe/some/path/to/Mnesia.company"` - это путь к директории, где будет храниться база данных.

## 17.7 The Table Viewer

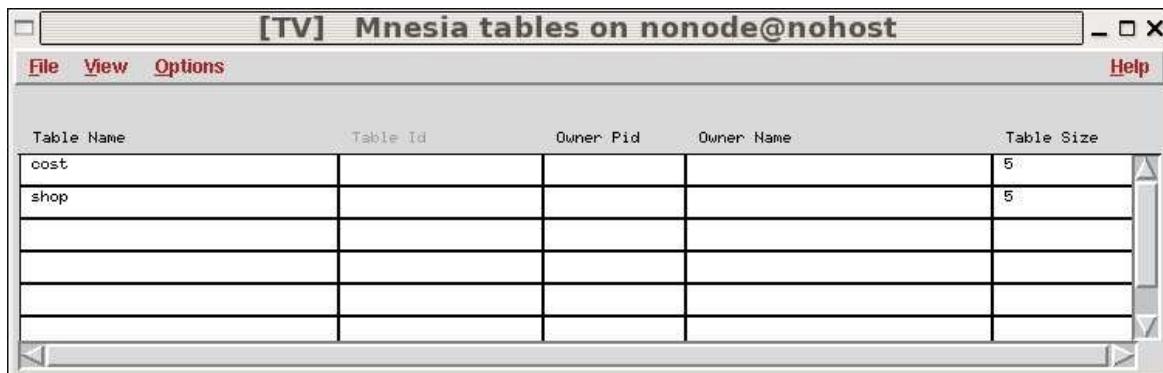


Рисунок 17.1: Начальный экран просмотрщика таблицы

Просмотрщик таблиц представляет собой GUI для просмотра таблиц Mnesia, а также таблиц ETS. Команда `tv:start()` запускает `table viewer`. Вы увидите начальный экран наподобие того, что приведен на рисунке 17.1. Чтобы увидеть таблицы Mnesia, вам нужно выбрать в меню `View > Tab`. На рисунке 17.2 мы видим, как `table viewer` показывает нам таблицу `shop` (на следующей странице).

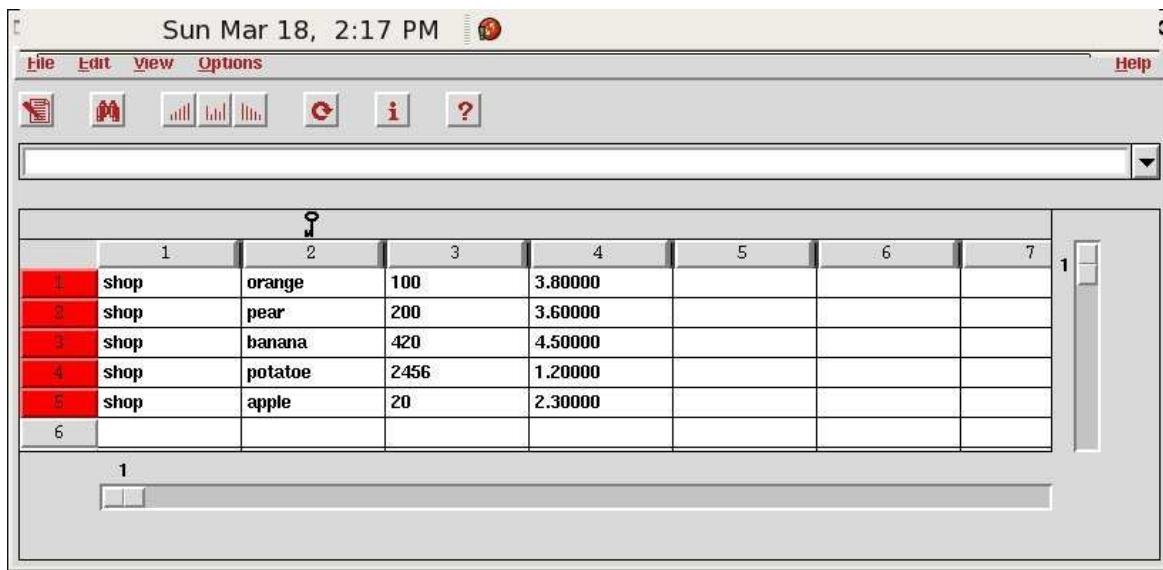


Рисунок 17.2: Просмотрщик таблицы

## 17.8 Куда копать?

Я надеюсь, мне удалось пробудить ваш аппетит к Mnesia. Mnesia является очень мощной СУБД. Она работает в промышленном использовании в ряде весьма требовательных телекоммуникационных систем, выпускаемых компанией Ericsson с 1998 года.

Поскольку вы читаете книгу об Эрланге, а не о СУБД Mnesia, мне придётся ограничиться лишь несколькими примерами наиболее типовых способов использования Mnesia. Приёмы, которые я продемонстрировал в этой главе, я использую в работе сам. Я бы не сказал, что использую (и понимаю) сильно больше того, что я уже показал вам. Но при помощи того, что я показал вам, вы можете делать довольно много, и создавать довольно развитые приложения.

Перечислю главные области, которые я не включил в эту главу:

- *Резервирование и восстановление*: у Mnesia есть ряд опций для настройки операций резервного копирования, подходящих для различных способов восстановления.
- "грязные" операции: Mnesia может выполнять ряд "грязных" операций (`dirty_read`, `dirty_write`, ...). Это операции, которые выполняются в обход транзакционного контекста. Это довольно опасные операции, которыми, впрочем, можно пользоваться, если ваше приложение является однопоточным, или при некоторых других особых обстоятельствах. "Грязные" операции используются в целях повышения эффективности.

- Таблицы SNMP: Mnesia предоставляет встроенный тип таблиц SNMP. Это делает реализацию систем управления SNMP довольно простой.

Исчерпывающий справочник по СУБД Mnesia - это Mnesia User's Guide, его можно найти на главном сайте дистрибутивов Erlang (см Приложение C, на стр. \_\_\_\_).

Дополнительно, поддиректория `examples`, включенная в дистрибутив Mnesia (`/usr/local/lib/erlang/lib/mnesia-X.Y.Z/examples` на моей машине) содержит ряд примеров использования Mnesia.

## 17.9 Listings

Скачать [test\\_mnesia.erl](#)

```
%% ---
%% Excerpted from "Programming Erlang",
%% published by The Pragmatic Bookshelf.
%% Copyrights apply to this code. It may not be used to create training
%% courses, books, articles, and the like. Contact us if you are in doubt.
%% We make no guarantees that this code is fit for any purpose.
%% Visit http://www.pragmaticprogrammer.com/titles/jaerlang for more book
%% ---
-module(test_mnesia).
-import(lists, [foreach/2]).
-compile(export_all).

%% IMPORTANT: The next line must be included
%% if we want to call qlc:q(...)

-include_lib("stdlib/include/qlc.hrl").

-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).

-record(design, {id, plan}).

do_this_once() ->
 mnesia:create_schema([node()]),
 mnesia:start(),
 mnesia:create_table(shop, [{attributes, record_info(fields, shop)}]),
 mnesia:create_table(cost, [{attributes, record_info(fields, cost)}]),
 mnesia:create_table(design, [{attributes, record_info(fields, design)}])
 mnesia:stop().

start() ->
 mnesia:start(),
 mnesia:wait_for_tables([shop, cost, design], 20000).
```

```

%% SQL equivalent
%% SELECT * FROM shop;

demo(select_shop) ->
 do qlc:q([X || X <- mnesia:table(shop)]));

%% SQL equivalent
%% SELECT item, quantity FROM shop;

demo(select_some) ->
 do qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)]));

%% SQL equivalent
%% SELECT shop.item FROM shop
%% WHERE shop.quantity < 250;

demo(reorder) ->
 do qlc:q([X#shop.item || X <- mnesia:table(shop),
 X#shop.quantity < 250
]);

%% SQL equivalent
%% SELECT shop.item
%% FROM shop, cost
%% WHERE shop.item = cost.name
%% AND cost.price < 2
%% AND shop.quantity < 250

demo(join) ->
 do qlc:q([X#shop.item || X <- mnesia:table(shop),
 X#shop.quantity < 250,
 Y <- mnesia:table(cost),
 X#shop.item =:= Y#cost.name,
 Y#cost.price < 2
]);

do(Q) ->
 F = fun() -> qlc:e(Q) end,
 {atomic, Val} = mnesia:transaction(F),
 Val.

example_tables() ->
 [% The shop table
 {shop, apple, 20, 2.3},

```

```

{shop, orange, 100, 3.8},
{shop, pear, 200, 3.6},
{shop, banana, 420, 4.5},
{shop, potato, 2456, 1.2},
%% The cost table
{cost, apple, 1.5},
{cost, orange, 2.4},
{cost, pear, 2.2},
{cost, banana, 1.5},
{cost, potato, 0.6}
].
add_shop_item(Name, Quantity, Cost) ->
Row = #shop{item=Name, quantity=Quantity, cost=Cost},
F = fun() ->
 mnesia:write(Row)
end,
mnesia:transaction(F).

remove_shop_item(Item) ->
Oid = {shop, Item},
F = fun() ->
 mnesia:delete(Oid)
end,
mnesia:transaction(F).

farmer(Nwant) ->
%% Nwant = Number of oranges the farmer wants to buy
F = fun() ->
 %% find the number of apples
 [Apple] = mnesia:read({shop,apple}),
 Napples = Apple#shop.quantity,
 Apple1 = Apple#shop{quantity = Napples + 2*Nwant},
 %% update the database
 mnesia:write(Apple1),
 %% find the number of oranges
 [Orange] = mnesia:read({shop,orange}),
 NOranges = Orange#shop.quantity,
 if
 NOranges >= Nwant ->
 N1 = NOranges - Nwant,
 Orange1 = Orange#shop{quantity=N1},
 %% update the database
 mnesia:write(Orange1);
 true ->

```

```

 %% Oops -- not enough oranges
 mnesia:abort(oranges)
 end
end,
mnesia:transaction(F).

reset_tables() ->
 mnesia:clear_table(shop),
 mnesia:clear_table(cost),
 F = fun() ->
 foreach(fun mnesia:write/1, example_tables())
 end,
 mnesia:transaction(F).

add_plans() ->
 D1 = #design{id = {joe,1},
 plan = {circle,10}},
 D2 = #design{id = fred,
 plan = {rectangle,10,5}},
 D3 = #design{id = {jane,{house,23}},
 plan = {house,
 [{floor,1,
 [{doors,3},
 {windows,12},
 {rooms,5}]},
 {floor,2,
 [{doors,2},
 {rooms,4},
 {windows,15}]}]}},
 F = fun() ->
 mnesia:write(D1),
 mnesia:write(D2),
 mnesia:write(D3)
 end,
 mnesia:transaction(F).

get_plan(PlanId) ->
 F = fun() -> mnesia:read({design, PlanId}) end,
 mnesia:transaction(F).

```

1. SQL - это очень известный язык, используемый для доступа к реляционным СУБД.
2. На самом деле, не столь удивительно, что SQL и обработчики списков похожи.  
Обе этих вещи основаны на математической теории множеств.

---

## Глава 18. Создание системы с использованием OTP

В этой главе мы будем создавать систему, выполняющую функции сервера в интернет-компании. Наша компания имеет два объекта продажи: простые числа и области ? услуги вычисления площади?. Покупатели могут купить простое число у нас или мы вычислим область геометрического объекта для них. Я думаю, что наша компания имеет огромный потенциал.

Мы создадим два сервера: один будет генерировать простые числа, а второй вычислять площадь. Чтобы сделать это, мы будем использовать `gen_server`, о котором мы говорили в разделе 16.2, "Начнем с `gen_server`" на странице 301.

Когда мы создаем систему, мы должны думать об ошибках, которые могут возникнуть. Хотя мы тщательно тестируем свое приложение, некоторые ошибки могут ускользнуть из поля зрения. Так что будем предполагать, что один из наших серверов может иметь фатальную ошибку, которая обрушит наш сервер. На самом деле, мы специально совершим ошибку, в одном из серверов, которая будет приводить к аварии.

Для обнаружения факта обрушения сервера нам необходимо иметь соответствующий механизм, чтобы определить, что случилась авария и перезапустить сервер. Для этого мы используем идею дерева супервизоров. Мы создадим супервизора, который будет следить за нашими серверами и перезапускать их при авариях.

Конечно же, если сервер потерпел аварию, мы захотим знать причины аварии, чтобы в дальнейшем устранить обнаруженные проблемы. Для протоколирования всех ошибок мы будем использовать регистратор ошибок OTP (OTP error logger). Мы покажем, как настраивать регистратор ошибок и как генерировать отчет об ошибках по журналу ошибок.

При вычислении простых чисел, в частности больших простых чисел, наш процессор может перегреться. Для предотвращения перегрева нам потребуется включать мощный вентилятор. Чтобы сделать это, нам нужно подумать о системе оповещения - тревогах ?алармах?. Мы будем использовать подсистему обработки событий OTP для генерирования и обработки тревог ?алармов?.

Все эти задачи (создание сервера, надзор за сервером, регистрация ошибок и определение тревог) являются типичными проблемами, которые должны быть решены в любой системе промышленного масштаба. В общем, даже если наша компания имеет довольно мутную перспективу, мы сможем использовать эту архитектуру в других системах. На самом деле, такая архитектура используется в ряде успешных коммерческих компаний.

В итоге, когда все заработает, мы упакуем весь наш код в единое OTP приложение. Вкратце, это специализированная группировка всех частей задачи, которая позволяет системе OTP запускать, управлять и останавливать задачу.

Порядок, в котором изложен материал несколько замысловат и имеет обратные зависимости между различными частями. Регистрация ошибок представляет собой особый случай управления событиями. Тревоги - это просто события, а сам регистратор ошибок - это контролируемый процесс, хотя процесс супервизора и может вызывать функции регистратора ошибок.

Я попытаюсь все это упорядочить и представить части в некотором осмысленном порядке. Итак, мы будем делать следующее:

1. Рассмотрим идеи использования типичного обработчика событий.
2. Увидим как работает регистратор ошибок.
3. Добавим управление тревогами.
4. Напишем два сервера.
5. Создадим дерево надзора и добавим в него наши серверы.
6. Упакуем всё в единое приложение.

---

## 18.1 Типичная обработка событий

Событие - это когда что-нибудь происходит, что-нибудь заслуживающее внимания программиста, который думает о том, кто и что должен делать в данном случае.

Когда мы программируем и происходит что-нибудь заметное, мы просто отправляем сообщение о событии зарегистрированному процессу. Что-то вроде этого:

```
RegProcName ! {event, E}
```

`E` - это событие (любой Эрланг-элемент (term)). `RegProcName` - имя зарегистрированного процесса.

Нам не нужно заботиться о том, что происходит с сообщением, когда мы его отправили. Мы просто выполнили работу и сообщили о том, что что-то случилось.

Теперь переключим наше внимание на процесс приёма сообщений о событиях. Этот процесс называется "обработчик событий". Простейший возможный обработчик событий - это "ничего не делающий" обработчик. Когда он принимает сообщение `{event, X}`, он ничего не делает с этим событием; просто отбрасывает его в сторону.

Вот наша первая попытка создания программы типичного обработчика событий:

[Скачать event\\_handler.erl](#)

```
-module(event_handler).
-export([make/1, add_handler/2, event/2]).

%% make a new event handler called Name
%% the handler function is noop -- so we do nothing with the event
make(Name) ->
 register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).

add_handler(Name, Fun) -> Name ! {add, Fun}.

%% generate an event
event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->
 receive
 {add, Fun1} ->
 my_handler(Fun1);
 {event, Any} ->
 (catch Fun(Any)),
 my_handler(Fun)
 end.

no_op(_) -> void.
```

API обработчика событий следующий:

```
event_handler:make(Name)
```

Приготовить "ничего не делающий" обработчик называемый Name (атом). Это то место, куда будут направляться события.

```
event_handler:event(Name, X)
```

Отправить событие `X` обработчику `Name`.

```
event_handler:add_handler(Name, Fun)
```

Добавить обработчик `Fun` к обработчику событий `Name`. Когда происходит событие `X`, обработчик выполнит `Fun(X)`.

Теперь создадим обработчик и сгенерируем ошибку:

```
1> event_handler:make(errors).
true
2> event_handler:event(errors, hi).
{event,hi}
```

Ничего особенного не произойдет, потому что мы не подключили модуль обратных вызовов к этому обработчику.

Чтобы получить обработчик событий, который делает что-нибудь осмысленное, необходимо написать для него модуль обратных вызовов и подключить этот модуль к обработчику:

Скачать [motor\\_controller.erl](#)

```
-module(motor_controller).
-export([add_event_handler/0]).

add_event_handler() ->
 event_handler:add_handler(errors, fun controller/1).

controller(too_hot) ->
 io:format("Turn off the motor\~n");

controller(X) ->
 io:format("\~w ignored event: \~p\~n" ,[?MODULE, X]).
```

Скомпилируем этот код и подключим к обработчику:

```
3> c(motor_controller).
{ok,motor_controller}
4> motor_controller:add_event_handler().
{add,\#Fun<motor_controller.0.99476749>}
```

Теперь, когда события будут отправлены обработчику, они будут обработаны функцией `motor_controller:controller/1`:

```
5> event_handler:event(errors, cool).
motor_controller ignored event: cool
{event,cool}
6> event_handler:event(errors, too_hot).
Turn off the motor
{event,too_hot}
```

И в чём же смысл проделанной работы? Во первых, мы задали имя, на которое будут отправляться события. В данном случае, это зарегистрированный процесс `errors`. Затем, мы определили протокол отправки событий зарегистрированному процессу. Но мы ничего не сказали о том, что происходит с событиями, которые получает этот процесс. На самом деле, всё что случается будет обработано в функции `noOp(X)`. В конце мы подключим другой обработчик событий, но об этом позже.

### Очень позднее связывание с "изменением ваших мыслей"

Предположим, что мы пишем функцию, которая скрывает конструкцию `event_handler:event` от программиста. Например, мы пишем следующее:

[Скачать lib\\_misc.erl](#)

```
too_hot() ->
 event_handler:event(errors, too_hot).
```

В этом случае мы говорим программисту вызывать `lib_misc:too_hot()` в своем коде, когда дела пойдут плохо. В большинстве языков программирования вызов функции `too_hot` был бы статически или динамически прилинкован в код программы. Так как вызов прилинкован, значит он выполняет фиксированную работу зависящую от кода. Если позднее изменится наше понимание и мы решим изменить что-нибудь, то это будет не простой путь изменения нашей системы.

Подход Эрланга к обработке событий абсолютно другой. Он позволяет отделить генерацию событий от обработки событий. Мы можем изменить обработку в любое время, просто передав новую функцию обработки в обработчик событий. Ничего не линкуется статически, и каждый обработчик может быть изменен тогда, когда вам это потребуется.

Используя такой механизм, мы можем построить систему *меняющуюся со временем* и не требующую остановки для замены кода.

*Примечание:* Это не "позднее связывание" - это "ОЧЕНЬ позднее связывание, дающее возможность думать так или иначе".

Возможно, вы немного запутались. Почему мы говорим об обработчиках событий? Ключевой момент повествования в том, что обработчик событий предоставляет нам инфраструктуру, в которую мы можем внедрять свои обработчики.

Инфраструктура регистратора ошибок строится из шаблона обработчика событий. Мы можем устанавливать различные обработчики в регистраторе ошибок для достижения различных целей.

## 18.2 Регистратор ошибок

Система OTP строится на настраиваемых регистрациях ошибок. Регистратор ошибок можно рассматривать с трех точек зрения. С точки зрения *программиста* - это вызовы функций позволяющие вести журнал ошибок. Точка зрения *конфигурации* - это то, как регистратор ошибок сохраняет данные. Точка зрения *отчетов* - это анализ ошибок после того как они случились. Мы рассмотрим каждую из точек зрения.

### Журналирование/Протоколирование ошибок

Что касается программиста, API регистрация ошибок достаточно прост. Вот он:

```
@spec error_logger:error_msg(String) -> ok
```

Отправить сообщение об ошибке регистратору ошибок.

```
1> error_logger:error_msg("An error has occurred\n").
=ERROR REPORT==== 28-Mar-2007::10:46:28 ===
An error has occurred
ok
```

```
@spec error_logger:error_msg(Format, Data) -> ok
```

Отправить сообщение об ошибке регистратору ошибок. Аргументы такие же как и для `io:format(Format, Data)`.

```
2> error_logger:error_msg("\~s, an error has occurred\n", ["Joe"]).
=ERROR REPORT==== 28-Mar-2007::10:47:09 ===
Joe, an error has occurred
ok
```

```
@spec error_logger:error_report(Report) -> ok
```

Отправить стандартный отчет об ошибке регистратору ошибок.

- `@type Report = [{Tag, Data} | term() | string() | term()]`
- `@type Tag = term()`
- `@type Data = term()`

```
error_logger:error_report([{tag1,data1},a_term,{tag2,data2}]).
```

```
=ERROR REPORT==== 28-Mar-2007::10:51:51 ===
tag1: data1
a_term
tag2: data
```

Это только небольшая часть доступного API. Обсуждение деталей не очень интересно. В наших программах мы будем использовать только `error_msg`. Полное описание можно посмотреть на страницах руководства по `error_logger`.

## Настройка регистратора ошибок

Существует много способов настроить регистратор ошибок. Мы можем видеть все ошибки в окне оболочки Эрланга (это режим по-умолчанию, специально настраивать не требуется). Мы можем записывать все ошибки попадающие в окно оболочки в один отформатированный файл. И, наконец, мы можем создать кольцевой ?циклический? журнал ошибок. Можете думать о кольцевом журнале как о большом кольцевом буфере, который содержит сообщения, выдаваемые регистратором ошибок. Новые сообщения записываются в конец журнала, а когда журнал полон, то записи из начала журнала удаляются.

Кольцевые журналы используются очень часто. Вам решать как много файлов журналов использовать и насколько они будут большими, а система сама позаботится об удалении старых и создании новых файлов в кольцевом буфере. Вы можете задать подходящий размер файла, чтобы сохранить в нем записи за несколько дней, этого обычно достаточно в большинстве случаев.

## Стандарные регистраторы ошибок

Когда мы запускаем Эрланг, мы можем использовать аргумент `boot`:

```
$ erl -boot start_clean
```

Такой запуск обеспечит окружение для разработки программ. Будет поддерживаться только простая регистрация ошибок. (Команда `erl` без аргумента `boot` эквивалентна команде `erl -boot start_clean`)

```
$ erl -boot start_sasl
```

Такой запуск обеспечит окружение для запуска системы готовой к эксплуатации. Библиотеки поддержки системной архитектуры (SASL - System Architecture Support Libraries) позаботятся о регистрации ошибок, о перегрузках системы и так далее.

Настройку журналов лучше всего делать из файлов настроек, потому что вряд ли кто-нибудьпомнит все аргументы регистратора. Далее мы рассмотрим как работает

система по-умолчанию и увидим четыре конфигурации, которые меняют поведение регистратора.

## SASL без настройки

Вот что происходит, когда мы запускаем SASL без файла настроек:

```
$ erl -boot start_sasl
Erlang (BEAM) emulator version 5.5.3 [async-threads:0] ...
=PROGRESS REPORT==== 27-Mar-2007::11:49:12 ===
supervisor: {local,sasl_safe_sup}
started: [{pid,<0.32.0>},
 {name,alarm_handler},
 {mfa,{alarm_handler,start_link,[[]]},restart_type,permanent},
 {shutdown,2000},
 {child_type,worker}]
...
... many lines removed ...
Eshell V5.5.3 (abort with ^G)
```

Сейчас мы вызовем одну из конструкций `error_logger` для отчета об ошибке:

```
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error
ok
```

Заметим, что отчет об ошибке отобразился в оболочке Эрланга. Вывод отчетов об ошибках зависит от настроек регистратора ошибок.

## Управление регистратором

Регистратор ошибок предоставляет несколько типов отчетов:

### Отчеты супервизора

Отчеты о том, что OTP супервизор запускает или останавливает подчинённые процессы (мы поговорим о супервизорах в разделе 18.5 "Дерево надзора", на странице 351).

### Отчеты о выполнении

Отчеты о запуске или остановке супрвизора.

## Отчеты об авариях

Отчеты об остановке выполнения с сообщением о причине отказа помимо `normal` или `shutdown`.

Эти три типа отчетов обрабатываются автоматически и не требуют вмешательства программиста.

Мы можем дополнительно вызвать конкретную конструкцию модуля `error_handler`, чтобы обработать все три типа отчетов. Это позволит нам использовать сообщения об ошибках, предупреждения и сообщения информационного характера. Три этих термина ничего не означают; воспринимайте их как теги, позволяющие программисту различать природу элементов в журнале ошибок.

Позже, когда журнал ошибок будет проанализирован, эти теги помогут нам решить какие из элементов журнала исследовать. Когда мы настраиваем регистратор ошибок, мы можем указать, что требуется сохранять только ошибки, а все остальные элементы игнорировать. Теперь давайте напишем файл настроек `elog1.config` для настройки регистратора ошибок:

[Скачать `elog1.config`](#)

```
%% no tty
[{sasl, [
 {sasl_error_logger, false}
]}].
```

Если мы запустим систему с этим файлом настроек, то будем получать только сообщения об ошибках, мы не получим сообщений о ходе выполнения и прочих. Все эти сообщения будут выводиться только в окно оболочки Эрланга.

```
$ erl -boot start_sasl -config elog1
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error
ok
```

## Текстовый файл и оболочка (shell)

Следующий файл настроек выдаёт список ошибок в окно оболочки Эрланга и дублирует все сообщения из оболочки в файл:

[Скачать `elog2.config`](#)

```
%> single text file - minimal tty
[{sasl, [
 %% All reports go to this file
 {sasl_error_logger, {file, "/home/joe/error_logs/THELOG" }}]}
]}].
```

Для проверки мы запустим Эрланг, сгенерируем сообщение об ошибке и посмотрим результат в файле:

```
$ erl -boot start_sasl -config elog2
1> error_logger:error_msg("This is an error\\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error ok
```

Если мы посмотрим файл `/home/joe/error_logs/THELOG`, в начале файла мы найдем следующие строки:

```
=PROGRESS REPORT==== 28-Mar-2007::11:30:55 ===
supervisor: {local,sasl_safe_sup}
started: [{pid,<0.34.0>},
 {name,alarm_handler},
 {mfa,{alarm_handler,start_link,[]}},
 {restart_type,permanent},
 {shutdown,2000},
 {child_type,worker}]
...
...
```

## Кольцевой журнал и оболочка

Эта конфигурация дает нам возможность выводить все ошибки в оболочку Эрланга плюс дублирование всего вывода оболочки в кольцевой журнальный файл. Такой вариант наиболее часто используется на практике.

[Скачать elog3.config](#)

```
%> rotating log and minimal tty
[{sasl, [
 {sasl_error_logger, false},
 %% задать параметры кольцевого журнала
 %% директория с файлом журнала
 {error_logger_mf_dir,"/home/joe/error_logs" },
 %% # кол-во байт выделенное для журнала
 {error_logger_mf_maxbytes,10485760}, % 10 MB
 %% максимальное кол-во файлов-журналов
```

```
{error_logger_mf_maxfiles, 10}
]}].

$erl -boot start_sasl -config elog3
1> error_logger:error_msg("This is an error\\n").
=ERROR REPORT==== 28-Mar-2007::11:36:19 ===
This is an error
false
```

При запуске системы все ошибки будут направляться в файл кольцевого журнала. Позже в этой главе мы рассмотрим как извлекать эти ошибки из файла-журнала.

## Продуктивная среда (ПРОДАКШЕН)

В продуктивной среде нам, на самом деле, интересны только отчеты об ошибках, а не о процессе выполнения или какая-либо информация, поэтому заставим регистратор отчитываться только об ошибках. Без этих настроек систему будут перегружать информационные отчеты и отчеты о процессе исполнения.

[Скачать elog4.config](#)

```
% rotating log and errors
[{sasl, [
 %% minimise shell error logging
 {sasl_error_logger, false},
 %% only report errors
 {errlog_type, error},
 %% define the parameters of the rotating log
 %% the log file directory
 {error_logger_mf_dir, "/home/joe/error_logs" },
 %% \# bytes per logfile
 {error_logger_mf_maxbytes, 10485760}, % 10 MB
 %% maximum number of
 {error_logger_mf_maxfiles, 10}
]}].
```

В результате запуска получим нечто похожее на предыдущий пример. С той лишь разницей, что регистрироваться будут только ошибки.

## Анализируем ошибки

Чтение журнала ошибок входит в обязанности модуля `rb`. Этот модуль имеет чрезвычайно простой интерфейс.

```
1> rb:help().
```

```

Report Browser Tool - usage

rb:start() - start the rb_server with default options
rb:start(Options) - where Options is a list of:
 {start_log, FileName}
 - default: standard_io
 {max, MaxNoOfReports}
 - MaxNoOfReports should be an integer or 'all'
 - default: all
...
... many lines omitted ...
...

```

Запустим браузер отчетов и скажем ему сколько записей из журнала читать (в данном случае последние двадцать):

```

2> rb:start([{max,20}]).
rb: reading report...done.
3> rb:list().
No Type Process Date Time
== ====
11 progress <0.29.0> 2007-03-28 11:34:31
10 progress <0.29.0> 2007-03-28 11:34:31
9 progress <0.29.0> 2007-03-28 11:34:31
8 progress <0.29.0> 2007-03-28 11:34:31
7 progress <0.22.0> 2007-03-28 11:34:31
6 progress <0.29.0> 2007-03-28 11:35:53
5 progress <0.29.0> 2007-03-28 11:35:53
4 progress <0.29.0> 2007-03-28 11:35:53
3 progress <0.29.0> 2007-03-28 11:35:53
2 progress <0.22.0> 2007-03-28 11:35:53
1 error <0.23.0> 2007-03-28 11:36:19
ok
> rb:show(1).

ERROR REPORT <0.40.0> 2007-03-28 11:36:19
=====
This is an error
ok

```

Для того чтобы найти конкретную ошибку мы можем использовать такую команду как rb:grep(RegExp), при её использовании найдется то, что описано в регулярном выражении RegExp. Я не хочу углубляться в то, как анализировать журналы ошибок.

Лучше потратьте время и поинтересуйтесь модулем `rb` и все увидите сами. Замечу, что на самом деле вам никогда не потребуется удалять журналы ошибок, точный механизм кольцевых журналов в конце концов сам удалит старые записи.

Если вам требуется оставить все сообщения об ошибках, вы можете задать интервалы и удалять информацию по мере необходимости.

## 18.3 Управление тревогами

Когда мы пишем наше приложение, нам требуется только одна тревога - будем реагировать только тогда, когда начнет перегреваться процессор, потому что мы вычисляем гигантские простые числа (помните, мы создали компанию по продаже простых чисел). Вот теперь-то мы и будем использовать настоящий OTP-шный обработчик тревог (и не простой как в начале главы).

Обработчик тревог это модуль обратных вызовов OTP для поведения `gen_event`. Вот его код:

Скачать [my\\_alarm\\_handler.erl](#)

```
-module(my_alarm_handler).
-behaviour(gen_event).

%% gen_event callbacks
-export([init/1, handle_event/2, handle_call/2,
 handle_info/2, terminate/2]).

%% init(Args) must return {ok, State}
init(Args) ->
 io:format("/**/ my_alarm_handler init:\n~p\n", [Args]),
 {ok, #{}}.

handle_event({set_alarm, tooHot}, N) ->
 error_logger:error_msg("/**/ Tell the Engineer to turn on the fan~n"),
 {ok, N+1};
handle_event({clear_alarm, tooHot}, N) ->
 error_logger:error_msg("/**/ Danger over. Turn off the fan~n"),
 {ok, N};
handle_event(Event, N) ->
 io:format("/**/ unmatched event:\n~p\n", [Event]),
 {ok, N}.

handle_call(_Request, N) -> Reply = N, {ok, N, N}.
```

```
handle_info(_Info, N) -> {ok, N}.

terminate(_Reason, _N) -> ok.
```

Этот код очень похож на код обратных вызовов `gen_server`, который мы видели раньше в разделе 16.3, "Что же происходит когда мы вызываем сервер?", на странице 306. Интересующей нас конструкцией является `handle_event(Event, State)`. Она возвращает `{ok, NewState}`. `Event` - это кортеж имеющий форму `{EventType, EventArg}`, где `EventType` это атом `set_event` или `clear_event`, а `EventArg` - это пользовательские аргументы. Чуть позже мы рассмотрим как генерируются такие события.

А теперь позабавимся. Мы запустим систему, сгенерируем тревогу, установим обработчик тревог, сгенерируем новую тревогу, и так далее:

```
$ erl -boot start_sasl -config elog3
1> alarm_handler:set_alarm(tooHot).
ok
=INFO REPORT==== 28-Mar-2007::14:20:06 ===
alarm_handler: {set,toohot}

2> gen_event:swap_handler(alarm_handler,
 {alarm_handler, swap},
 {my_alarm_handler, xyz}).

*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}
3> alarm_handler:set_alarm(tooHot).
ok
=ERROR REPORT==== 28-Mar-2007::14:22:19 ===
*** Tell the Engineer to turn on the fan
4> alarm_handler:clear_alarm(tooHot).
ok
=ERROR REPORT==== 28-Mar-2007::14:22:39 ===
*** Danger over. Turn off the fan
```

## Что же здесь происходит?

- Мы запустили Эрланг с `-boot start_sasl`. Когда мы сделали это, мы получили стандартный обработчик тревог. Когда мы устанавливаем или очищаем тревогу, то ничего не происходит. Это простой "ничего не делающий" обработчик событий, мы такие рассматривали раньше.
- Когда мы установили тревогу (строка 1), мы просто получили информационный

отчет. Здесь нет специальной обработки тревог.

3. Мы установили свой обработчик тревог (строка 2). Аргумент в `my_alarm_handler` (`xyz`) не имеет особого значения; синтаксис требует какое-нибудь значение, но поскольку нам не требуются значения, мы просто используем атом `xuz`, мы сможем увидеть этот аргумент при выводе на консоль.  
Строка `** my_alarm_handler_init: ...` напечатана из нашего модуля обратных вызовов.
4. Мы установили и очистили тревогу `tooHot` (строки 3 и 4). Это отработал наш обработчик тревог. Мы можем проверить, прочитав вывод на консоли.

## Чтение журнала

Давайте вернёмся обратно к регистратору ошибок и посмотрим, что там происходит:

```
1> rb:start([{max,20}]).
rb: reading report...done.
2> rb:list().
No Type Process Date Time
== === ===== === ===
...
3 info_report <0.29.0> 2007-03-28 14:20:06
2 error <0.29.0> 2007-03-28 14:22:19
1 error <0.29.0> 2007-03-28 14:22:39
3> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 14:22:39
=====*** Danger over. Turn off the fan
ok
4> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 14:22:19
=====*** Tell the Engineer to turn on the fan
```

Итак, здесь мы видим как работает механизм регистрация ошибок.

На практике мы должны были бы убедиться, что журнал ошибок достаточно велик для хранения данных за несколько дней или даже недель. Каждые несколько дней (или недель) мы бы проверяли журнал на предмет ошибок.

Примечание: Модуль `rb` содержит функции для выбора ошибок указанного типа и

извлечения этих ошибок в файл. В результате процесс анализа ошибок может быть полностью автоматизирован.

## 18.4 Серверные приложения

Наше приложение состоит из двух серверов: сервер простых чисел и сервер рассчёта площади. Рассмотрим сервер простых чисел. Он написан с использованием поведения gen\_server (см. раздел 16.2 "Начинаем с gen\_server" на стр. 301). Замечу, что он включает в себя обработку тревог которую мы разработали в предыдущем разделе.

### Сервер простых чисел

[Скачать prime\\_server.erl](#)

```
-module(prime_server).
-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
 terminate/2, code_change/3]).

start_link() ->
 gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->
 %% 20000 is a timeout (ms)
 gen_server:call(?MODULE, {prime, N}, 20000).

init([]) ->
 %% Note we must set trap_exit = true if we
 %% want terminate/2 to be called when the application
 %% is stopped
 process_flag(trap_exit, true),
 io:format("\~p starting\~n" ,[?MODULE]),
 {ok, []}.

handle_call({prime, K}, _From, N) ->
 {reply, make_new_prime(K), N+1}.

handle_cast(_Msg, N) -> {noreply, N}.
```

```

handle_info(_Info, N) -> {noreply, N}.

terminate(_Reason, _N) ->
 io:format("\~p stopping\~n" ,[?MODULE]),
 ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

make_new_prime(K) ->
 if
 K > 100 ->
 alarm_handler:set_alarm(tooHot),
 N = lib_primes:make_prime(K),
 alarm_handler:clear_alarm(tooHot),
 N;
 true ->
 lib_primes:make_prime(K)
 end.

```

## Сервер площи

Теперь рассмотрим сервер площи. Он так же построен на поведении `gen_server`. Заметьте, написание сервера очень быстрый процесс. Когда я писал этот пример, я просто скопировал код из сервера простых чисел и вставил его в новый сервер. Все заняло несколько минут.

Сервер площи не является идеальной программой и содержит преднамеренную ошибку (сможете ее найти?). Мой не очень коварный план - это заставить сервер рухнуть, чтобы быть перестрахованным супервизором. А потом получить отчет обо всех ошибках в журнале ошибок.

[Скачать prime\\_server.erl](#)

```

-module(area_server).
-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
 terminate/2, code_change/3]).

start_link() ->
 gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

```

```

area(Thing) ->
 gen_server:call(?MODULE, {area, Thing}).

init([]) ->
 %% Note we must set trap_exit = true if we
 %% want terminate/2 to be called when the application
 %% is stopped

process_flag(trap_exit, true),
io:format("~p starting~n" ,[?MODULE]),
{ok, 0}.

handle_call({area, Thing}, _From, N) ->
 {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N) -> {noreply, N}.

handle_info(_Info, N) -> {noreply, N}.

terminate(_Reason, _N) ->
 io:format("~p stopping~n" ,[?MODULE]),
 ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

compute_area({square, X}) -> X*X;

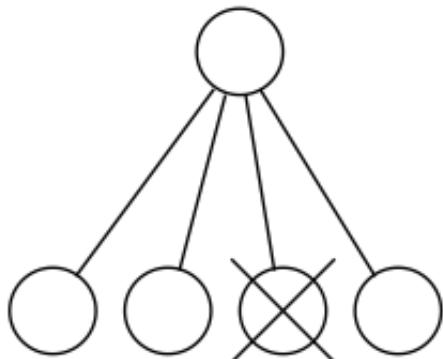
compute_area({rectonge, X, Y}) -> X*Y.

```

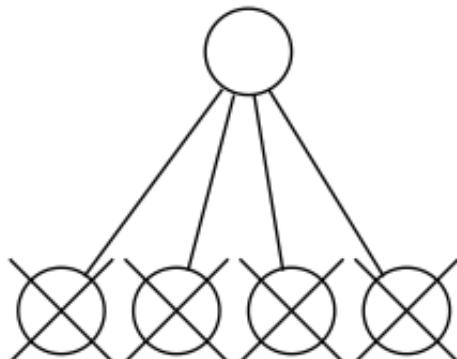
надзор `one_for_one` Если один процесс рухнет, он будет перезапущен

надзор `all_for_one` Если один процесс рухнет, все процессы будут прерваны и перезапущены

one\_for\_one supervision  
If one process crashes, it is restarted



all\_for\_one supervision  
If one process crashes, all are terminated  
and then restarted



18.1 Два вида дерева надзора

## 18.5 Дерево надзора

Дерево надзора - это дерево процессов. Самые верхние процессы (супервизоры) в дереве наблюдают за нижними (рабочими) процессами в дереве и перезапускают нижние процессы, если те аварийно завершаются. Два вида дерева надзора вы можете увидеть на рисунке 18.1.

*One-for-one* дерево надзора

В надзоре *one-for-one*, если один процесс рухнул, то супервизор рестартует только этот процесс.

*All-for-one* дерево надзора

В надзоре *all-for-one*, если любой из процессов рухнет, то все поднадзорные процессы будут уничтожены (вызовом функции `terminate/2` в соответствующем модуле обратных вызовов). Затем все рабочие процессы будут рестартованы.

Супервизоры создаются с использованием OTP поведения *supervisor*. Это поведение описывается в специальном модуле обратных вызовов, который содержит стратегию надзора и правила запуска отдельных рабочих процессов в дереве надзора. Дерево надзора определяется функцией следующего вида:

```
init(...) ->
 {ok, {RestartStrategy, MaxRestarts, Time},
 [Worker1, Worker2, ...]}.
```

Здесь RestartStrategy это один из атомов one\_for\_one или all\_for\_one. MaxRestarts и Time указывают на "частоту перезапуска". Если супервизор перезапускает процессы большее число раз, чем указано в MaxRestarts за Time секунд, то работа супервизора будет прервана. Это делается для того, чтобы остановить бесконечный цикл перезапуска процессов, если они содержат ошибки и останавливаются из-за них.

`Worker1`, `Worker2` и т.д. это кортеж описывающий как запускать каждый из рабочих процессов. Мы увидим, как это выглядит уже скоро.

Теперь давайте вернемся к нашей компании и создадим дерево надзора.

Для начала, думаю, нам надо выбрать имя для нашей компании. Пусть будет `sellaprime`. Задача супервизора `sellaprime` - это конечно же держать всегда запущенными сервер простых чисел и сервер площади. Для этого напишем уже другой модуль обратных вызовов, теперь для `gen_supervisor`.

Вот этот модуль:

[Скачать sellaprime\\_supervisor.erl](#)

```
-module(sellaprime_supervisor).
-behaviour(supervisor). % see erl -man supervisor

-export([start/0, start_in_shell_for_testing/0, start_link/1,
 init/1]).

start() ->
 spawn(fun() ->
 supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])
 end).

start_in_shell_for_testing() ->
 {ok, Pid} = supervisor:start_link({local,?MODULE},
 ?MODULE, _Arg = []),
 unlink(Pid).

start_link(Args) ->
 supervisor:start_link({local,?MODULE}, ?MODULE, Args).

init([]) ->
 %% Install my personal error handler
 gen_event:swap_handler(alarm_handler,
 {alarm_handler, swap},
 {my_alarm_handler, xyz}),
```

```
{ok, [{one_for_one, 3, 10},
 [{tag1,
 {area_server, start_link, []},
 permanent,
 10000,
 worker,
 [area_server]},
 {tag2,
 {prime_server, start_link, []},
 permanent,
 10000,
 worker,
 [prime_server]}}
]}].
```

Самая важная часть - это структура данных возвращаемая функцией `init/1`:

[Скачать sellaprime\\_supervisor.erl](#)

```
{ok, [{one_for_one, 3, 10},
 [{tag1,
 {area_server, start_link, []},
 permanent,
 10000,
 worker,
 [area_server]},
 {tag2,
 {prime_server, start_link, []},
 permanent,
 10000,
 worker,
 [prime_server]}}
]}].
```

Эта структура данных определяет стратегию надзора. Мы говорили о стратегии надзора и частоте перезапуска выше. Сейчас осталось дать определение для сервера областей и сервера простых чисел.

Определение для Worker процессов имеет следующий вид:

```
{Tag, {Mod, Func, ArgList},
 Restart,
 Shutdown,
 Type,
 [Mod1]}
```

Что же обозначают все эти аргументы?

### Tag

Атом, который будет использоваться для ссылки на рабочий процесс в дальнейшем (если потребуется).

### {Mod, Func, ArgList}

Определение функции, которую супервизор будет использовать для запуска рабочего процесса. Оно используется как аргумент при вызове `apply(Mod, Fun, ArgList)`.

### Restart = permanent | transient | temporary

`permanent` - процесс будет перезапускаться всегда. `transient` - процесс будет перезапущен только, если получено ненормальное значение при выходе. `temporary` - процесс запускается только один раз и не перезапускается.

### Shutdown

Время остановки. Это максимально разрешенное время для остановки рабочего процесса. Если время остановки процесса будет превышено, то процесс просто будет убит. (Возможны и другие значения - см. руководство по Супервизору)

### Type = worker | supervisor

Тип надзираемого процесса. Мы можем сконструировать дерево надзора над супервизорами, добавляя процесс супервизора вместо рабочего процесса.

### [Mod1]

Это имя модуля обратных вызовов, если дочерний процесс имеет поведение `supervisor` или `gen_server` (Возможны и другие значения - см. руководство по Супервизору)

## 18.6 Запуск системы

Теперь мы готовы первый раз запустить нашу компанию. Мы вернулись. Кто хочет купить первое простое число?

Давайте запустим систему:

```
$ erl -boot start_sasl -config elog3
1> sellaprime_supervisor:start_in_shell_for_testing().
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
```

Теперь сделаем правильный запрос:

```
2> area_server:area({square,10}).
100
```

Сейчас сделаем неправильный запрос:

```
3> area_server:area({rectangle,10,20}).
area_server stopping
=ERROR REPORT==== 28-Mar-2007::15:15:54 ===
** Generic server area_server terminating
** Last message in was {area,{rectangle,10,20}}
```

## Действительно ли работает стратегия надзора?

Эрланг был разработан для программирования отказоустойчивых систем.

Первоначальная разработка была сделана в Лаборатории Вычислительной Техники Шведской компании Эрикссон. С тех пор группа OTP вела разработку с помощью десятков сотрудников компании. Используя `gen_server`, `gen_supervisor` и другие поведения Эрланга строились системы с надежностью 99.9999999% (тут девять девяток). При правильном использовании, механизм обработки ошибок может помочь сделать вашу программу работающей вечно (ну, или почти вечно). Регистратор ошибок, описанный здесь, работает уже в течение нескольких лет в реальных, живых продуктах.

```
** When Server state == 1
** Reason for termination ==
** {{function_clause,
 [{area_server,compute_area,[{rectangle,10,20}]}],
 [{area_server,handle_call,3},
 {gen_server,handle_msg,6},
 {proc_lib,init_p,5}]}}
area_server starting
** exited: {{function_clause,
 [{area_server,compute_area,[{rectangle,10,20}]}],
 [{area_server,handle_call,3},
 {gen_server,handle_msg,6},
 {proc_lib,init_p,5}]},
 {gen_server,call,
```

```
[area_server,{area,{rectangle,10,20}}]}]} **
```

Упс - что же тут случилось? Сервер площади рухнул; мы умышленно допустили в коде ошибку. Авария была обнаружена супервизором и сервер был перезапущен. Все это запротоколировал регистратор ошибок.

После аварии, все вернулось к нормальному состоянию, как и должно было. Давайте сейчас сделаем правильный запрос:

```
4> area_server:area({square,25}).
625
```

У нас все опять работает. Теперь давайте сгенерируем маленькое простое число:

```
5> prime_server:new_prime(20).
Generating a 20 digit prime
37864328602551726491
```

А теперь сгенерируем большое простое число:

```
6> prime_server:new_prime(120).
Generating a 120 digit prime
=ERROR REPORT==== 28-Mar-2007::15:22:17 ===
*** Tell the Engineer to turn on the fan
.....

=ERROR REPORT==== 28-Mar-2007::15:22:20 ===
*** Danger over. Turn off the fan
765525474077993399589034417231006593110007130279318737419683
288059079481951097205184294443332300308877493399942800723107
```

Теперь у нас работоспособная система. Если на сервере случится авария, то он автоматически будет перезапущен, а регистратор ошибок проинформирует нас об этом.

Сейчас давайте рассмотрим журнал ошибок:

```
1> rb:start([{max,20}]).
rb: reading report...done.
rb: reading report...done.
{ok,<0.53.0>}
2> rb:list().
No Type Process Date Time
== === ===== == ===
```

```

20 progress <0.29.0> 2007-03-28 15:05:15
19 progress <0.22.0> 2007-03-28 15:05:15
18 progress <0.23.0> 2007-03-28 15:05:21
17 supervisor_report <0.23.0> 2007-03-28 15:05:21
16 error <0.23.0> 2007-03-28 15:07:07
15 error <0.23.0> 2007-03-28 15:07:23
14 error <0.23.0> 2007-03-28 15:07:41
13 progress <0.29.0> 2007-03-28 15:15:07
12 progress <0.29.0> 2007-03-28 15:15:07
11 progress <0.29.0> 2007-03-28 15:15:07
10 progress <0.29.0> 2007-03-28 15:15:07
9 progress <0.22.0> 2007-03-28 15:15:07
8 progress <0.23.0> 2007-03-28 15:15:13
7 progress <0.23.0> 2007-03-28 15:15:13
6 error <0.23.0> 2007-03-28 15:15:54
5 crash_report area_server 2007-03-28 15:15:54
4 supervisor_report <0.23.0> 2007-03-28 15:15:54
3 progress <0.23.0> 2007-03-28 15:15:54
2 error <0.29.0> 2007-03-28 15:22:17
1 error <0.29.0> 2007-03-28 15:22:20

```

Что-то тут не так. У нас есть отчет об аварии сервера областей. Как узнать, что случилось (если бы мы не знали об этом)?

```

9> rb:show(5).
CRASH REPORT <0.43.0> 2007-03-28 15:15:54
=====
Crashing process
pid <0.43.0>
registed_name area_server
error_info
{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
 {area_server,handle_call,3},
 {gen_server,handle_msg,6},
 {proc_lib,init_p,5}]}

initial_call
{gen,init_it,
 [gen_server,
 <0.42.0>,
 <0.42.0>,
 {local,area_server},
 area_server,
 [],
 []]}

ancestors [sellaprime_supervisor,<0.40.0>]

```

|            |            |
|------------|------------|
| messages   | □          |
| links      | [<0.42.0>] |
| dictionary | □          |
| trap_exit  | false      |
| status     | running    |
| heap_size  | 233        |
| stack_size | 21         |
| reductions | 199        |
| ok         |            |

Распечатка `{function_clause, compute_area, ...}` отображает точное место в программе сервера где произошла авария. Это должно помочь легко локализовать и исправить ошибку. Давайте перейдем к рассмотрению следующих ошибок:

```
10> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 15:22:17
=====
*** Tell the Engineer to turn on the fan
```

и

```
10> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 15:22:20
=====
*** Danger over. Turn off the fan
```

Это предупреждения нашей системы охлаждения при вычислении очень больших простых чисел!

## 18.7 Приложение

Мы почти закончили. Всё что нам осталось сделать - это написать файл с расширением .app который будет содержать информацию о нашем приложении:

[Скачать sellaprime.app](#)

```
%% This is the application resource file (.app file) for the 'base'
%% application.

{application, sellaprime,
 [{description, "The Prime Number Shop" },
```

```

{vsn, "1.0" },
{modules, [sellaprime_app, sellaprime_supervisor, area_server,
 prime_server, lib_primes, my_alarm_handler]},
{registered,[area_server, prime_server, sellaprime_super]},
{applications, [kernel,stdlib]},
{mod, {sellaprime_app,[]}},
{start_phases, []}
]}.

```

Теперь нам потребуется написать модуль обратных вызовов с именем модуля из предыдущего примера:

[Скачать sellaprime\\_app.erl](#)

```

-module(sellaprime_app).
-behaviour(application).
-export([start/2, stop/1]).

%%-----%
%% Function: start(Type, StartArgs) -> {ok, Pid} |
%% {ok, Pid, State} |
%% {error, Reason}
%% Description: This function is called whenever an application
%% is started using application:start/1,2, and should start the
%% processes
%% of the application. If the application is structured according to the
%% OTP design principles as a supervision tree, this means starting the
%% top supervisor of the tree.
%%-----%

start(_Type, StartArgs) ->
 sellaprime_supervisor:start_link(StartArgs).

%%-----%
%% Function: stop(State) -> void()
%% Description: This function is called whenever an application
%% has stopped. It is intended to be the opposite of Module:start/2 and
%% should do any necessary cleaning up. The return value is ignored.
%%-----%

stop(_State) ->
 ok.

```

Здесь должны быть экспортированы функции start/2 и stop/1. Раз мы уже все это сделали, то теперь можем запустить наше приложение в оболочке Эрланга.

```

$ erl -boot start_sasl -config elog3
1> application:loaded_applications().
[{:kernel,"ERTS CXC 138 10","2.11.3"},
 {:stdlib,"ERTS CXC 138 10","1.14.3"},
 {:sasl,"SASL CXC 138 11","2.1.4"}]
2> application:load(sellaprime).
ok
3> application:loaded_applications().
[{:sellaprime,"The Prime Number Shop","1.0"},{},{},{}]
4> application:start(sellaprime).
*** my_alarm_handler init:{xyz,[alarm_handler,[]]}
area_server starting
prime_server starting
ok
5> application:stop(sellaprime).
prime_server stopping
area_server stopping

=INFO REPORT==== 2-Apr-2007::19:34:44 ===
application: sellaprime
exited: stopped
type: temporary
ok
6> application:unload(sellaprime).
ok

7> application:loaded_applications().
[{:kernel,"ERTS CXC 138 10","2.11.4"},
 {:stdlib,"ERTS CXC 138 10","1.14.4"},
 {:sasl,"SASL CXC 138 11","2.1.5"}]

```

Вот теперь это вполне оперившееся приложение. Во второй строке мы загрузили приложение; этот вызов загружает весь код, но не запускает приложение. В четвертой строке мы запустили приложение, а в пятой строке остановили его. Заметьте, все видно в распечатке, когда приложения запускаются и останавливаются, соответствующие функции сервера простых чисел и сервера площади были вызваны. В шестой строке мы выгрузили приложение. Все модули приложения были удалены из памяти.

Когда мы делаем полноценную систему, используя OTP, мы упаковываем её в приложение. Это даёт нам универсальный метод запуска, остановки и управления

приложением.

Заметьте, когда мы используем `init:stop()` для завершения работы системы, то все приложения будут завершены так, как это принято. Это правило хорошего тона.

```
$ erl -boot start_sasl -config elog3
1> application:start(sellaprime).
*** my_alarm_handler init:{xyz,[alarm_handler,[]]}
area_server starting
prime_server starting
ok
2> init:stop().
ok
prime_server stopping
area_server stopping
$
```

Две строки следующие за командой номер 2 получены из сервера площади и сервера простых чисел, они показывают нам, что был вызван метод `terminate/2` из модуля обратных вызовов `gen_server`.

## 18.8 Организация файловой системы

Я до сих пор ничего не упоминал об организации файловой системы. Это сделано специально - моя цель озадачивать вас проблемами по одной.

Структурированное OTP приложение, обычно, содержит файлы соответствующие различным частям приложения в строго определенных местах. Это не требование; так как все нужные файлы могут быть найдены во время исполнения, но это не потребуется, если все файлы лежат в нужных местах.

Все описанные в этой книге демонстрационные файлы располагаются в одной директории. Это простейшие примеры, и сделано это во избежание проблем с путями поиска и с взаимодействием между различными программами.

Основные файлы, используемые в компании sellaprime, следующие:

| <i>File</i>                   | <i>Content</i>                                                         |
|-------------------------------|------------------------------------------------------------------------|
| <code>area_server.erl</code>  | Сервер областей - модуль обратных вызовов <code>gen_server</code>      |
| <code>prime_server.erl</code> | Сервер простых чисел - модуль обратных вызовов <code>gen_server</code> |

|                                       |                                                            |
|---------------------------------------|------------------------------------------------------------|
| <code>sellaprim_supervisor.erl</code> | Модуль обратных вызовов Супервизора                        |
| <code>sellaprim_app.erl</code>        | Модуль обратных вызовов Приложения                         |
| <code>my_alar_handler.erl</code>      | Модуль обратных вызовов Событий для <code>gen_event</code> |
| <code>sellaprime.app</code>           | Спецификация приложения                                    |
| <code>elog4.config</code>             | Файл настроек Регистратора ошибок                          |

Для рассмотрения того как используются эти файлы и модули нам нужно рассмотреть последовательность событий, происходящих, когда приложение стартует:

1. Мы запускаем систему следующими командами:

```
$ erl -boot start_sasl -config elog4.config
1> application:start(sellaprime).
...

```

Файл `sellaprime.app` должен находиться в корневом каталоге из которого запускается Эрланг или в подкаталоге этого каталога.

В этом случае контроллер приложения сможет обнаружить `{mod, ...}`, объявленный в `sellaprime.app`. Он содержит имя контроллера приложения. И конечно же это модуль `sellaprime_app`.

2. Вызывается обратный вызов `sellaprime_app:start/2`.
3. `sellaprime_app:start/2` вызывает `sellaprime_supervisor:start_link/2`, который, в свою очередь, запускает супервизор `sellaprime`.
4. Вызывается обратный вызов супервизора `sellaprime_supervisor:init/1` - он устанавливает обработчик ошибок и возвращает спецификацию надзора. Спецификация надзора сообщает как запускать сервер площади и сервер простых чисел.
5. `sellaprime` супервизор запускает сервер площади и сервер простых чисел. Они реализованы как модули обратных вызовов `gen_server`.

Останавливать все это очень просто. Вы просто вызываете `application:stop(sellaprime)` или `init:stop()`.

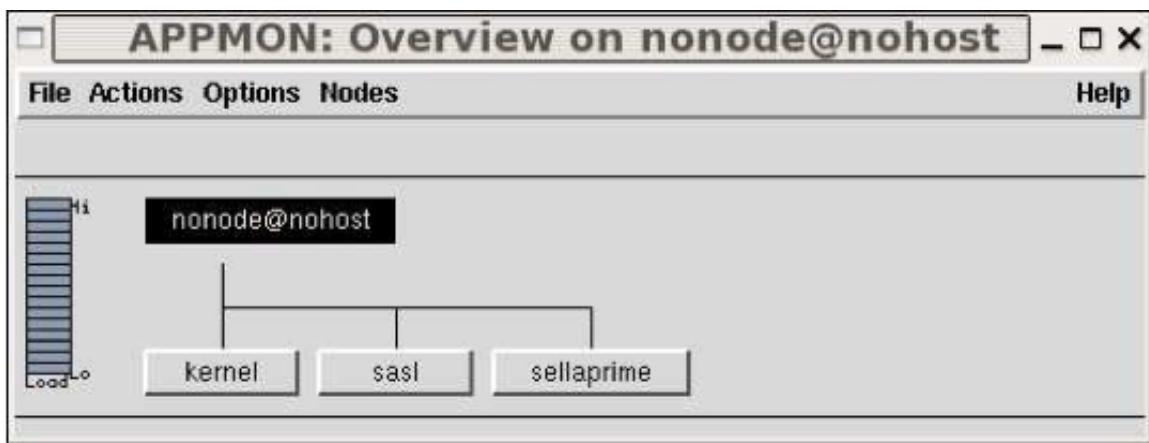
## 18.9 Монитор приложений

Монитор приложений - это программа с графическим интерфейсом (GUI) для

просмотра запущенных приложений. Команда `appmon:start()` запускает монитор приложений. Когда вы выполните эту команду, вы увидите окно, похожее на то, которое изображено на рисунке 18.2. Для того чтобы увидеть структуру приложения, вы должны кликнуть по одному из приложений. Монитор приложений для приложения `sellaprime` показан на рисунке 18.3.

## 18.10 Копаем глубже

Я пропустил довольно много подробностей, разъяснив только принципы. Вы сможете найти подробности на страницах руководства по `gen_event`, `error_logger`, `supervisor` и `application`.



18.2 Монитор приложений. Вид при запуске.

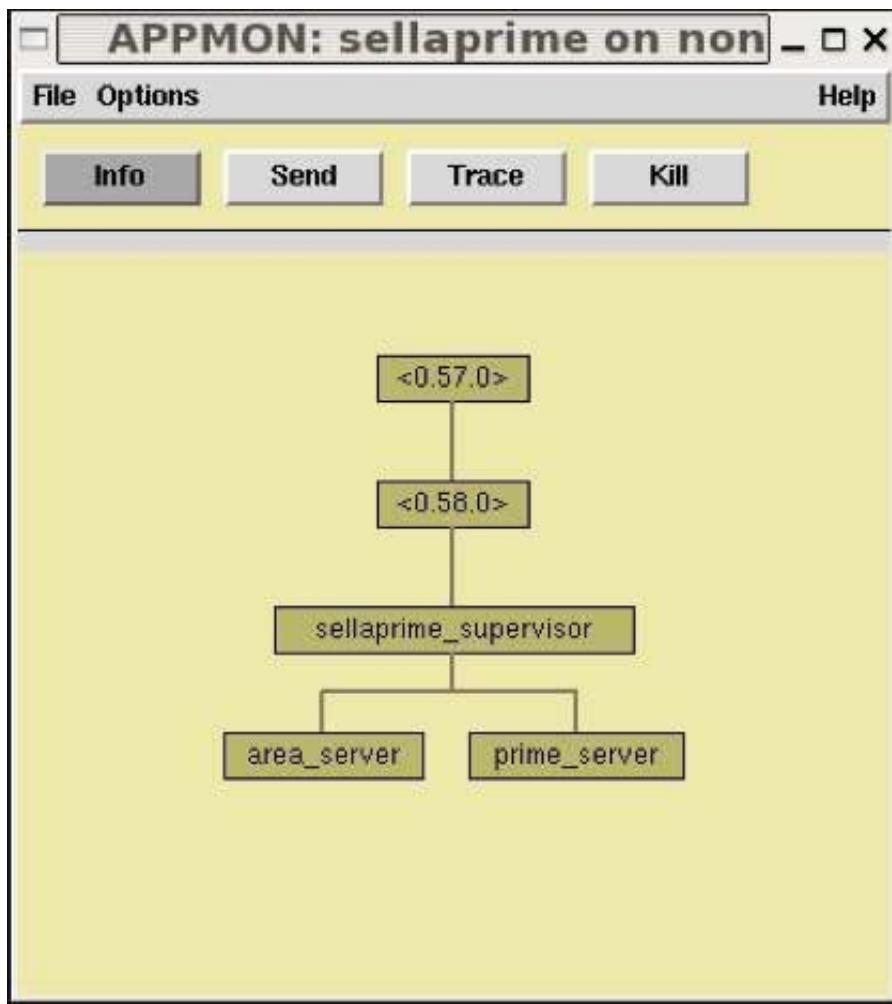


Рисунок 18.3 Приложение sellaprime

Следующие файлы содержат более подробную информацию о том как использовать OTP поведения:

[http://www.erlang.org/doc/pdf/design\\_principles.pdf](http://www.erlang.org/doc/pdf/design_principles.pdf) (страница 97) Gen servers, gen event, supervisors

[http://www.erlang.org/doc/pdf/system\\_principles.pdf](http://www.erlang.org/doc/pdf/system_principles.pdf) (страница 19) Как сделать boot файл

<http://www.erlang.org/doc/pdf/appmon.pdf> (страница 16) Монитор приложений

## 18.11 Как мы вычисляем простые числа?

Очень просто.

Скачать [lib\\_primes.erl](#)

```

%% make a prime with at least K decimal digits.
%% Here we use 'Bertrand's postulate.
%% Bertrands postulate is that for every N > 3,
%% there is a prime P satisfying N < P < 2N - 2
%% This was proved by Tchebychef in 1850
%% (Erdos improved this proof in 1932)

make_prime(1) ->
 lists:nth(random:uniform(5), [1,2,3,5,7]);
make_prime(K) when K > 0 ->
 new_seed(),
 N = make_random_int(K),
 if N > 3 ->
 io:format("Generating a ~w digit prime " ,[K]),
 MaxTries = N - 3,
 P1 = make_prime(MaxTries, N+1),
 io:format("~n" ,[]),
 P1;
 true ->
 make_prime(K)
 end.

make_prime(0, _) ->
 exit(impossible);
make_prime(K, P) ->
 io:format("." ,[]),
 case is_prime(P) of
 true -> P;
 false -> make_prime(K-1, P+1)
 end.

%% Fermat's little theorem says that if
%% N is a prime and if A < N then
%% A^N mod N = A

is_prime(D) ->
 new_seed(),
 is_prime(D, 100).

is_prime(D, Ntests) ->
 N = length(integer_to_list(D)) -1,
 is_prime(Ntests, D, N).

is_prime(0, _, _) -> true;

```

```

is_prime(Ntest, N, Len) ->
K = random:uniform(Len),
%% A is a random number less than N
A = make_random_int(K),
if
 A < N ->
 case lib_lin:pow(A,N,N) of
 A -> is_prime(Ntest-1,N,Len);
 _ -> false
 end;
 true ->
 is_prime(Ntest, N, Len)
end.
```

```

1> lib_primes:make_prime(500).
Generating a 500 digit prime

7910157269872010279090555971150961269085929213425082972662439

1259263140285528346132439701330792477109478603094497394696440

4399696758714374940531222422946966707622926139385002096578309

0625341667806032610122260234591813255557640283069288441151813

9110780200755706674647603551510515401742126738236731494195650

5578474497545252666718280976890401503018406521440650857349061

2139806789380943526673726726919066931697831336181114236228904

0186804287219807454619374005377766827105603689283818173007034

056505784153
```

1. Я называю это итератором, но если быть точным, то это оператор свёртки (fold)  
очень похожий на lists:foldl.[↪](#)
2. Взято с <http://www.dcs.shef.ac.uk/research/ilash/Moby/>[↪](#)
3. Другими аргументами являются `exports`, `imports` и `compile`.[↪](#)
4. Можно использовать аргумент `global` для того, чтобы он был доступен кластеру  
Эрланг-серверов).[↪](#)
5. Доступно на [http://www.erlang.org/doc/design\\_principles/users\\_guide.html](http://www.erlang.org/doc/design_principles/users_guide.html)[↪](#)
6. [http://www.erlang.org/doc/design\\_principles/users\\_guide.html](http://www.erlang.org/doc/design_principles/users_guide.html)[↪](#)