Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической
кибернетики и компьютерных наук

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ERLANG

КУРСОВАЯ РАБОТА

студента 2 курса 211 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Мирзоева Никиты Романовича

Научный руководитель
зав.каф. МКиКН, к. ф.-м. н. _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Erlang — это функциональный язык программирования и среда исполнения со встроенной поддержкой распределенных и параллельных вычислений, инструментами обеспечения безотказной работы и множеством других интересных свойств, позволяющих писать еще более надёжный и масштабируемый код, при всём этом оставаясь достаточно простым.

Задачи данной курсовой работы:

- ознакомление с синтаксисом языка;
- изучение основных особенностей языка Erlang;
- проведение сравнительной характеристики Erlang с другими языками программирования;
- применение на практике полученных знаний о языке.

1 История Erlang

Информация об истории создания и развития Erlang была взята из книги Francesco Cesarini и Simon Thompson «Erlang Programming» [1].

В середине 1980-х годов сотрудники компьютерной лаборатории Ericsson работали над задачей поиска максимально подходящего языка для создания телекоммуникационных приложений нового поколения. В течение двух лет Джо Армстронг, Роберт Вирдинг и Майкл Вилльямс под руководством Бъёрна Дакера занимались прототипированием телекоммуникационных приложений на различных языках программирования. В результате анализа исследований, они пришли к выводу, что ни один язык не может полностью решить поставленную задачу, и решили создать новый. На Erlang оказали влияние такие функциональные языки, как ML и Miranda, языки параллельных вычислений ADA, Modula и Chill и язык логического программирования Prolog. Способ обновления приложений был позаимствован у Smalltalk. Также в Erlang можно найти черты внутренних языков компании Ericsson EriPascal и PLEX.

Первая виртуальная машина, на которой работал Erlang, была написана на Prolog. Четыре года на развивающемся языке писались прототипы телекоммуникационных приложений. В результате чего и был сформирован Erlang в таком виде, в каком он представлен в настоящее время.

Первая виртуальная машина на C была написана Майклом Виллиамсом в 1991 году. Год спустя был начат первый коммерческий проект. В этом проекте разрабатывался сервер, предоставляющий пользователям радиотелефонов DECT возможность общаться на ходу внутри частных офисных сетей. Проект был успешно запущен в 1994 году. Опыт эксплуатации этого проекта был особенно важен для разработчиков Erlang. Они учли недостатки в релизе Erlang 1995 года.

Только после этого Erlang начали применять в разработке крупных приложений, в которых участвовали сотни программистов. Среди этих приложений — организация широкополосной передачи, сеть GPRS и коммутатор ATM. В ходе работы над этими проектами были разработаны библиотеки OTP (Open Telecom Platform), выпущенные в 1996 году. Они предоставляют набор стандартных средств для построения надёжных и устойчивых программ.

Знание истории Erlang важно для понимания принципов языка. Erlang отличается от многих языков общего назначения тем, что он изначально разра-

батывался для решения конкретных задач, связанных с созданием распределённых, устойчивых к отказам систем, работающих в режиме реального времени, которые приспособленны к выполнению большого числа задач параллельно. Однако приложения, подходящие под эти описания, могут применяться не только в телекоммуникации, но и в веб-службах, компьютерной телефонии, банковских системах, системах обмена сообщениями, интеграции корпоративных приложений и многих других. Этим объясняется всплеск интереса к Erlang.

В 1998 году Erlang стал свободно распространяемым языком. Он был выпущен под лицензией EPL, производной от лицензии Mozilla Public License. В январе 1999 года на сайте erlang.org [5] было зарегистрировано 36 000 посещений, десять лет спустя это число возросло до 2,8 миллиона. Эта динамика говорит о стремительном росте Erlang-сообщества, который вызван совокупностью таких факторов, как выпуск ряда успешных коммерческих, научных и открытых приложений, вирусный маркетинг, посты в блогах и книги. И всё это для решения трудных задач в области, для которой Erlang был изначально создан.

2 Область применения языка

Несмотря на то, что Erlang изначально создавался конкретно для производства в сфере телекоммуникаций, область применения языка всё больше и больше расширяется. Например, кластерные системы, облачные сервисы, системы мгновенного обмена сообщениями.

Erlang хорошо подходит для разработки веб-серверов. За счёт легковесных процессов и простой работы с двоичными данными ускоряется обработка запросов со стороны клиента. Функциональные возможности языка помогают правильно выстроить бизнес-логику приложения. Кэширование, встроенное в язык, делает быстрым доступ к сохранённым данным.

Язык Erlang применяли и применяют в своих проектах множество известных компаний:

- Ericsson
- Facebook
- WhatsApp
- GitHub
- Google
- Yandex
- Amazon
- Grid Dynamics
- Yahoo!
- Motorola
- TMobile
- Call of Duty
- World of Tanks

3 Особенности Erlang

Несмотря на то что Erlang сам по себе достаточно хорош, его преимущества становятся очевидными при совместном использовании Erlang с виртуальной машиной, библиотеками OTP и многими другими библиотеками.

Можно выделить следующие основные особенности языка Erlang:

— Сильная динамическая типизация
— Функциональная парадигма
— Параллельные вычисления и обмен сообщениями
— Горячая замена кода и применение в системах реального времени
— Распределённость и масштабируемость
— Надёжность

3.1 Сильная динамическая типизация

Одной из важнейших особенностей Erlang является неизменяемость переменных, пришедшая из функционального программирования. Таким образом реализуется модель вычислений без состояний. В языке отсутствует неявное приведение типов. Поэтому Erlang относится к языкам с сильной типизацией.

В Erlang не существует отдельных типов, поэтому для удобства сохранения различных данных (числа, атомы, корежи, функции и т.д.) в языке применяется динамическая типизация. Это свойство помогает создавать надёжные программы, устойчивые к возникновению непредвиденных ситуаций.

Языки с сильной динамической типизацией:

— Python
— Ruby
— Clojure
— Common Lisp
— Elixir

Языки со слабой статической типизацией:

— C
— C++

В следующем примере продемонстрировано создание и использование переменных, содержащих различные структуры данных (списки, кортежи, хеши) в языках с сильной динамической типизацией. За счёт неизменяемости

переменных 6 строка на Erlang не компилируется. В Python присутствует неизменяемость переменных, однако переменная pyNumber в строке 6 указывает на область памяти, отличную от той, в которой находится переменная из строки 3. В Ruby все переменные (в том числе и константы) изменяемые.

```
% Erlang

Number = 45.
List = [].
Tuple = { ok, 8080 }.
Number = 56. % Ошибка компиляции, нельзя изменить переменную Number
```

```
# Python

pyNumber = 235
pyList = []
pyTuple = ( 1, 2, 3 )
pyNumber = pyTuple[2] # Числа (и не только) в Python неизменяемые,
                # но присваивание другого значения переменным возможно,
                # однако pyNumber указывает теперь на другую область памяти
```

```
# Ruby

rb_number = 211
rb_array = [ 5, "hello", true ]
rb_hash = { 'color' => 'green', 'number' => 5 }
RB_CONST = 1001 # Константа
RB_CONST = 1002 # Константы в Ruby изменяемые
```

3.2 Функциональная парадигма

Erlang построен на принципах функциональной парадигмы программирования. Функциональные программирование — это декларативное программирование, то есть оно описывает результат, а не способ его достижения. Поэтому функциональное программирование основано на математических моделях решения задач.

Особенностью функциональных языков являются возможность обращаться с функциями, как со значениями любых других типов. В Erlang для функций определён специальный тип данных — `fun`. Значения этого типа могут быть переданы в функцию в качестве аргумента и сохранены в любой структуре

данных, например в кортеже или списке. Их можно отправлять в сообщениях, и даже более того — возвращать из других функций. В Erlang функция - не статический указатель на определённый объект, а значение, которое может быть создано динамически.

Функциональность в Erlang позволяет писать более короткие и надёжные программы. Далее описаны наиболее важные особенности, пришедшие из функционального программирования.

3.2.1 Функции высшего порядка

Функции высшего порядка позволяют писать краткие и обобщённые функции, с помощью которых можно параметризовать функцию некоторым поведением. В результате чего повышается не только компактность кода, но и его наглядность, такой код проще для написания и восприятия.

Когда функции рассматриваются, как функции высшего порядка, их можно принимать как аргумент, возвращать как результат или сохранять в переменные.

Далее приведен пример работы с функциями высшего порядка в Erlang и в JavaScript. В строках 13 и 14 описывается функция `summa`, которая складывает два переданных ей числа. В строках 7-11 описывается функция `makeOperation`, принимающая в виде параметров другую функцию и два числа. В строке 8 вызывается функция, переданная в `makeOperation` первым аргументом, результат которой записывается в переменную `Number`. В строках 9-11 происходит возвращение результата функции `makeOperation`. Причем результатом является анонимная функция с одним параметром `Number2`, вычисляющая сумму `Number2` и `Number`.

```
1 % Erlang
2
3 main() ->
4     FunNum = makeOperation( summa, 1, 2 ), % функция сохраняется в переменную
5     io:format( "~w", [ FunNum(3) ] ).
6
7 makeOperation( Func, First, Second ) -> % функция принимается как аргумент
8     Number = Func( First, Second ),
9     fun( Number2 ) ->
10         Number2 + Number
11     end. % функция возвращается как результат
```

```
12
13 summa( FirstNum, SecondNum ) ->
14     FirstNum + SecondNum.
```

```
1 // JavaScript
2
3 function main()
4 {
5     var funNum = makeOperation( summa, 1, 2 );
6     document.getElementById('result').innerHTML = funNum(3);
7 }
8
9 function makeOperation( func, first, second )
10 {
11     var number = func( first, second );
12     return function ( number2 )
13     {
14         return number2 + number;
15     }
16 }
17
18 var summa = function( first_num, second_num )
19 {
20     return first_num + second_num;
21 }
```

3.2.2 Сопоставление с образцом

Сопоставление с образцом — метод анализа и обработки структур данных, основанный на выполнении определённых инструкций в зависимости от совпадения исследуемого значения с тем или иным образцом.

В Erlang сопоставление с образцом используется для:

- присваивания значений переменным;
- управления порядком выполнения;
- извлечения значений из составных типов данных.

Сопоставление с образцом имеет вид:

`Pattern = Expression`

Выражение `Pattern` состоит из структуры данных, которая содержит переменные (связанные и несвязанные) и литералы (числа, атомы, строки).

Связанной называют переменную, которой уже присвоено значение, а несвязанной — переменную, которой значение ещё не было присвоено.

Также, сопоставление с образцом можно применять в объвлении функции.

В следующем примере используется сопоставление с образцом в описании функций, переводящие целые числа в атомы (Erlang) или строки (Scala, Haskell).

```
% Erlang

matchTest( 1 ) -> one;
matchTest( 2 ) -> two;
matchTest( _ ) -> many.
```

```
// Scala

def matchTest( x: Int ): String = x match
{
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
}
```

```
-- Haskell

matchTest 1 = "one"
matchTest 2 = "two"
matchTest _ = "many"
```

3.2.3 Генераторы списков

Генераторы списков позволяют одним выражением создавать, фильтровать и трансформировать списки. Значение выражения будет списком тех элементов, извлечённых из генераторов списков, для которых предикат вернул атом true.

В общем виде генераторы списков строятся следующим образом:

`[ Expression || Generators, Guards, ... ]`

`Expression` — Выражения, определяющие в каком виде элементы попадут в результат.

`Generators` — Генератор, имеющий вид: `Pattern <- List`, где `Pattern` — образец, который проходит сопоставление с элементами списка, возвращаемого выражением `List`. Символ `<-` имеет смысл принадлежности элемента списку.

Guards — Охранные выражения, помогающие произвести проверку на выполнимость установленных условий.

Генераторы списков помогают значительно сократить реализацию стандартных алгоритмов, например алгоритм быстрой сортировки. В функциональных языках Erlang и Haskell похожим образом можно написать данную сортировку в несколько строк. В том случае, когда в функцию передается пустой список, возвращается пустой список. Иначе из списка выбирается опорный элемент (самый первый) и относительного него сортируются элементы, которые меньше или больше его.

```
% Erlang
% с использованием генераторов списков

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

```
-- Haskell
-- с использованием генераторов списков

qsort [] = []
qsort ( x : xs ) = qsort [ y | y <- xs, y <= x ] ++ [x]
                ++ qsort [ y | y <- xs, y > x ]
```

Для сравнения приведена стандартная реализация алгоритма быстрой сортировки на языке C++.

```
// C++
// без использования генераторов списков

int partition (int *mas, int l, int r)
{
        int x = mas[l];
        int i = l;
        int j = r;
        while (true)
        {
                while (mas[i] < x)
                        i++;
```

```
13                  while (mas[j] > x)
14                          j--;
15                  if (i < j)
16                          swap(mas[i], mas[j]);
17                  else
18                          return j;
19          }
20  }
21
22  void quick_sort(int *mas, int left, int right)
23  {
24          if (left < right - 1)
25          {
26                  int q = partition(mas, left, right - 1);
27                  quick_sort(mas, left, q);
28                  quick_sort(mas, q + 1, right);
29          }
30  }
```

3.2.4 Хвостовая рекурсия

Функцию называют функией с хвостовой рекурсией, если она содержит вызов самой себя только в последнем выражении в каждом из её функциональных уравнений. Главным отличием обычной рекурсии от хвостовой является наличие локальных переменных. В функциях с хвостовой рекурсией эти переменные становятся параметрами.

Использование хвостовой рекурсии гарантирует выполнение «бесконечных» процессов при фиксированном расходе памяти.

В следующем примере реализовано вычисление N-ого числа Фибоначчи на Erlang и Scala двумя способами: в виде простой рекурсивной функции и с использованием функции с хостовой рекурсией.

```
1 % Erlang
2
3 % рекурсивная функция
4 fib( N ) when N < 2 -> N;
5 fib( N ) ->
6         fib( N - 1 ) + fib( N - 2 ).
7
8 % функция с хвостовой рекурсией
9 fibIter( Prev, Current, 0 ) ->
```

```
10          Current;
11 fibIter( Prev, Current, N ) ->
12          fibIter( Current, Prev + Current, N - 1 ).
13
14 fib( N ) when N < 2 -> N;
15 fib( N ) ->
16          fibIter( 1, 1, N - 2 ).
```

```
1 // Scala
2
3 // рекурсивная функция
4 public static int fib( int n )
5 {
6          if ( n > 1 ) return fib( n - 1 ) + fib( n - 2 );
7      else return n;
8 }
9
10 // функция с хвостовой рекурсией
11 public static int fib( int n )
12 {
13          if ( n > 1 ) return fibIter( 1, 1, n - 2 );
14      else return n;
15 }
16
17 private static int fibIter( int prev, int current, int n )
18 {
19          if ( n == 0 ) return current;
20          else return fibIter( current, prev + current, n - 1 );
21 }
```

3.2.5 Ленивые списки

Функциональное программирование позволяет использовать ленивые (отложенные) вычисления — операции, которые вычисляются только непосредственно по запросу в тот момент, когда они понадобятся.

В Erlang можно построить ленивые списки. Для этого нужно поместить в хвост списка функцию, которая будет возвращать следующую голову списка и рекурсивную функцию хвоста списка. При этом не нужно генерировать большой список, для того чтобы обойти его. Получение следующего значения списка происходит тогда, когда оно необходимо, уменьшая расход памяти.

Следующая функция на Erlang позволяет сгенерировать ленивый список.

```
% Erlang

next( Seq ) ->
        fun() -> [ Seq | next( Seq + 1 ) ] end.
```

Данный пример показывает, как с помощью ленивых вычислений строятся бесконечные списки в Haskell.

```
-- Haskell

inf_list = [1..]
```

3.3 Параллельные вычисления и обмен сообщениями

Параллелизм, основанный на легковесных процессах, стал основой успеха Erlang. Вместо использования потоков вычислений, которые обмениваются данными с помощью разделяемой памяти операционной системы, в Erlang для каждого процесса выделяется отдельная область памяти. В Erlang не возникает лишних зависимостей процессов, допущенных по невнимательности, которые могут привести к взаимным блокировкам или прочим проблемам. Каждый процесс обрабатывает только одно событие и заканчивается, когда обработка события завершена.

Процессы взаимодействуют друг с другом через передачу сообщений. Сообщение может содержать любое значение Erlang. Чаще всего сообщение содержит кортеж вида `{Result, Reason}`. Положительным свойством передачи сообщений является асинхронность. Как только сообщение отправлено, процесс продолжает выполняться. Время передачи сообщения пренебрежимо мало и не зависит от числа запущенных процессов. Сообщения извлекаются из почтового ящика процесса выборочно. Не обязательно обрабатывать сообщения в порядке их поступления. Это повышает надёжность приложений, особенно если приложение выполняется на сети из нескольких компьютеров и скорость передачи сообщения зависит от физических характеристик соединения.

Данный код демонстирует пример создания процессов на Erlang. В строках 3-4 описана функция `start`, которая для нужного количества элементов вызывает функцию `newProcess`. При помощи рекурсии внутри функции `newProcess` создаётся нужное количество процессов и информация о создании процесса отображается на экран. Для создания процесса в Erlang служит

функция spawn. 14 строка служит для передачи сообщения новому процессу. Строки 15-17 служат для приёма сообщений от других процессов.

```
% Erlang

start( Number ) ->
        newProccess( Number, self() ).

newProccess( 1, Pid ) ->
        io:format( "New proccess was starting.~n
                                Number of proccess: ~w~n", [ 1 ] ),
        Pid ! ok;

newProccess( Number, Pid ) ->
        NewPid = spawn( test_proc, newProccess, [ Number - 1, Pid ] ),
        io:format( "New proccess was starting.~n
                                Number of proccess: ~w~n", [ Number ] ),
        NewPid ! ok,
        receive ok ->
                ok
        end.
```

```
// Java

public static void main(String[] args) {
    for ( counter = 0; counter < count_of_threads; counter++ ) {
        new Thread() {
            public void run() {
                System.out.println("New proccess was starting.\n" +
                                + "Number of proccess: " + counter + "\n");
                Object new_obj = new Object();
                synchronized (new_obj) {
                    try {
                        new_obj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }
}
```

На следующих графиках (1 и 2) отображено сравнение по времени создания процессов на Erlang и потоков на Java в зависимости от их количества:

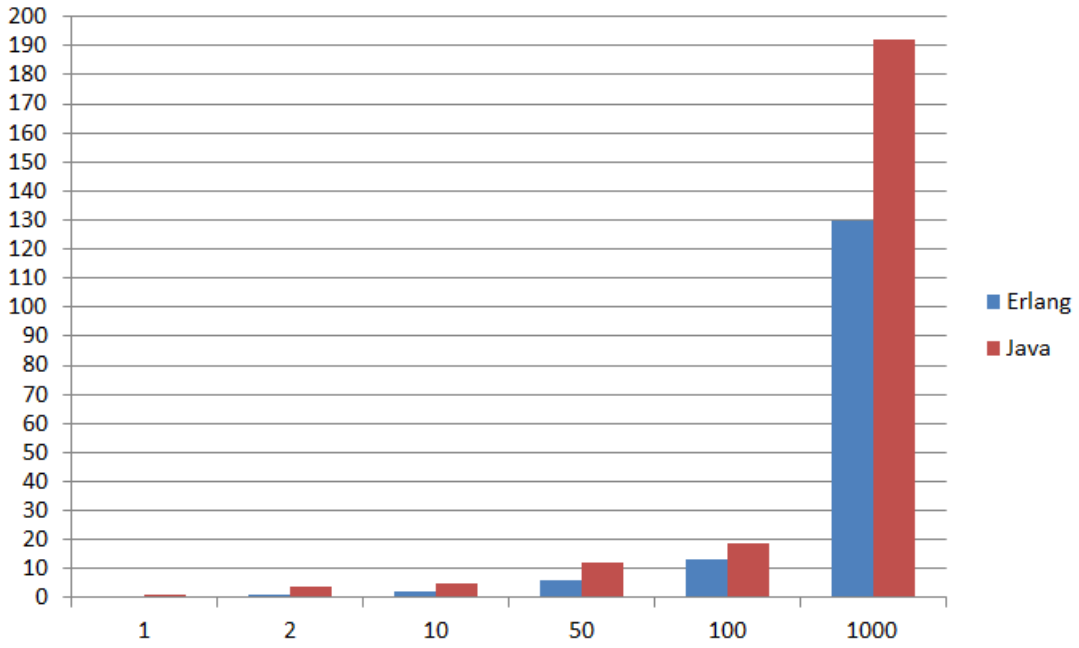


Рисунок 1 – Сравнение потоков на Java и процессов на Erlang

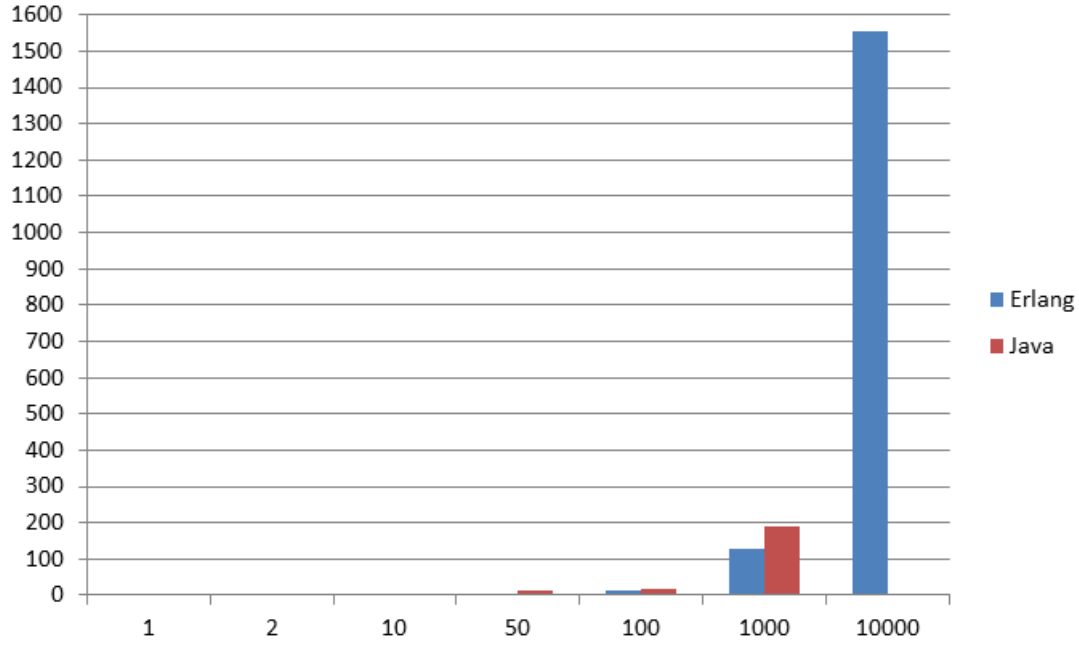


Рисунок 2 – Сравнение потоков на Java и процессов на Erlang

Нельзя создать 10000 потоков на Java, а на Erlang можно, причем это требует достаточно немного времени.

3.4 Горячая замена кода и применение в системах реального времени

Для систем, которые не могут быть остановлены для обновления кода, Erlang предлагает горячую замену кода. При этом в приложении могут одновременно работать старая и новая версии кода. Таким способом программное обеспечение на Erlang может быть модернизировано без простоев, а выявленные ошибки исправлены.

Несмотря на то что Erlang — высокоуровневый язык, его можно использовать и в системах псевдореального времени (soft real-time system). Управление памятью в Erlang производится сборщиком мусора для каждого процесса в отдельности. За счёт чего время отклика системы имеет порядок миллисекунд даже при управлении памятью с помощью сборщика мусора. Поэтому пропускная способность не снижается даже при высокой нагрузке.

3.5 Распределённость и масштабируемость

Инструменты для создания распределённых систем встроены в структуру языка. По умолчанию узлы обмениваются данными по протоколу TCP/IP. Они могут быть соединены в однородную сеть, в которой каждый из узлов не зависит от операционной системы, на которой он запущен. Можно создать полноценную сеть распределённых вычислений, если все узлы соединить в TCP/IP-сеть и правильно настроить сетевой экран.

Так как кластеры Erlang были разработаны для работы за сетевым экраном, безопасность достигается за счёт использования секретных куков с незначительными ограничениями прав доступа. При необходимости можно создавать сети Erlang, взаимодействующие между собой по безопасным протоколам SSL.

Программы Erlang имеют возможность легко масштабироваться за счёт простоты увеличения количества узлов, на которых располагается система. Так как Erlang владеет встроенными средствами для организации распределённых вычислений, решение таких задач, как распределение вычислительной нагрузки, кластеризация, добавление новых узлов или вычислительных средств и обеспечение надёжности передачи данных, не занимает много времени и усилий.

3.6 Надёжность

Erlang упрощает задачу создания надёжного отказоустойчивого приложения. Это возможно благодаря простому, но мощному механизму обработки ошибок и управления исключениями. На их основе были построены обобщённые библиотеки OTP (Open Telecom Platform). При испольховании OTP достаточно лишь программировать корректное поведение, обрабатывая исключения с помощью этих библиотек. Обычно при таком подходе код получается не только короче и проще, но и содержит меньшее количество ошибок.

OTP представляет следующие средства управления исключениями и построения структуры программы:

- Процессы в Erlang могут быть соединены так, что если один из них падает, то второй узнает об этом и либо исправит проблему, либо сам завершится.
- В OTP определены обобщённые поведения, такие как серверы, конечные автоматы и обработчики событий. Общие части для этих шаблонов уже запрограммированы, поэтому при создании программ, в которых используются эти поведения, нужно лишь добавить специфические детали реализации логики.
- Эти обобщённые процессы соединяются с процессом-наблюдателем. Единственная обязанность процесса-наблюдателя заключается в наблюдении за процессами и обработке завершения процессов. Процессы-наблюдатели могут следить как за рабочими процессами, так и за другими процессами-наблюдателями. Так, средствами OTP можно построить иерархию процессов-наблюдателей, называемую деревом наблюдателей 3.
- С помощью наблюдения за процессами и соединения процессов программисты могут сконцентрироваться на реализации корректного поведения, позволяя процессам падать. Отсутствие защитного программирования значительно облегчает задачу разработчика и делает код более понятным.

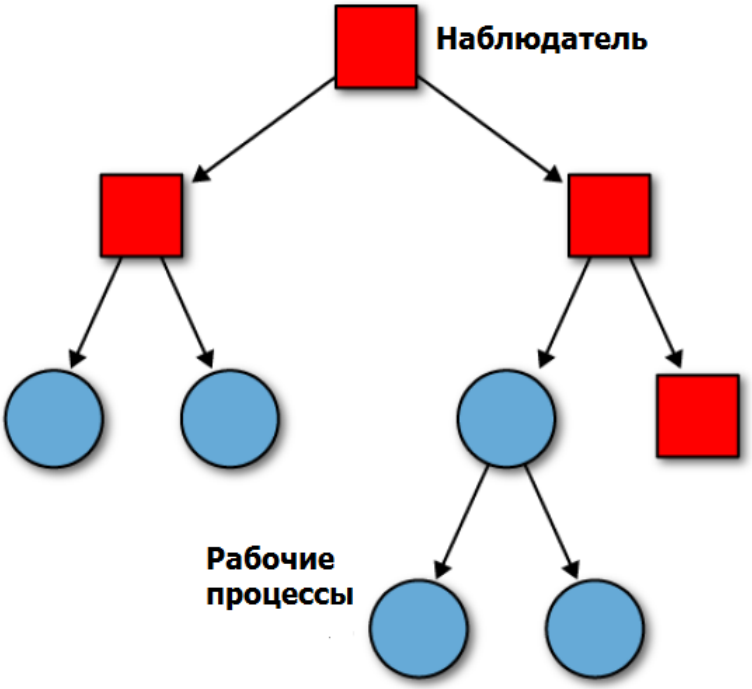


Рисунок 3 – Дерево наблюдателей

4 Практическое применение Erlang

На Erlang разработано множество программ и продуктов. В том числе серверы (Yaws, N2O, Cowboy, Chicago Boss) и СУБД (Mnesia, CouchDB). Для получения практических навыков программирования на Erlang я написал небольшой модуль для работы с Mnesia. Данный модуль помогает вести учет сотрудников различных отделов компании. В базу данных можно поместить информацию о названии отдела и о его сотрудниках. Информация о сотрудниках представляет собой кортеж из следующих данных:

— Имя сотрудника
— Фамилия сотрудника
— Заработная плата сотрудника

Функции, описанные в модуле позволяют безопасно добавлять и извлекать данные из БД.

Далее идут несколько примеров вызова функций, разработанного мной, модуля на Erlang.

Рисунок 4 – Запуск БД и добавление нового отдела

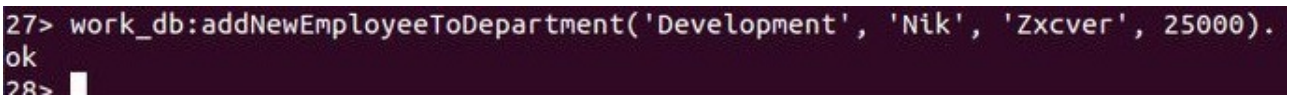
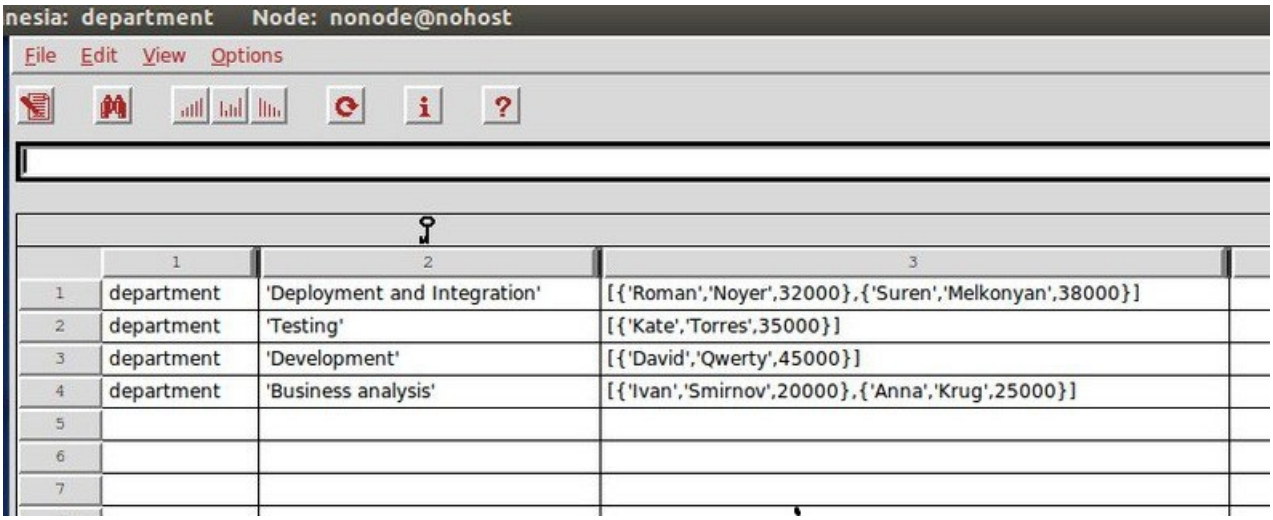


Рисунок 5 – Добавление нового сотрудника

Листинг программы приведен в приложении А.



	1	2	3
1	department	'Deployment and Integration'	[{'Roman','Noyer',32000},{'Suren','Melkonyan',38000}]
2	department	'Testing'	[{'Kate','Torres',35000}]
3	department	'Development'	[{'David','Qwerty',45000}]
4	department	'Business analysis'	[{'Ivan','Smirnov',20000},{'Anna','Krug',25000}]
5			
6			
7			

Рисунок 6 – Таблица БД

```
30> work_db:deleteOldEmployeeFromDepartment('Business analysis', 'Ivan', 'Smirnov', 20000).
ok.
31>
```

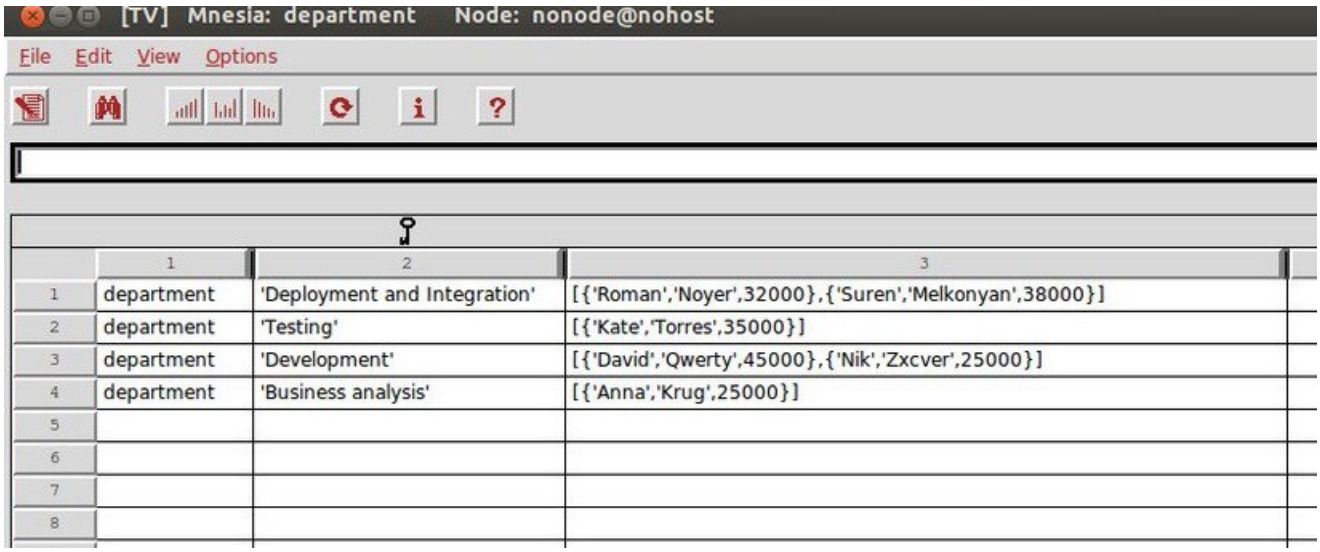
Рисунок 7 – Удаление сотрудника из базы

```
35> work_db:printDepartment('Development').
{'Development',[{'David','Qwerty',45000},
                {'Nik','Zxcver',25000}]}
36> work_db:printDepartment('Business analysis').
{'Business analysis',[{'Anna','Krug',25000}]}
```

Рисунок 8 – Получение информации из БД об отделе

```
41> work_db:delDepartment('Testing').
not_empty
```

Рисунок 9 – Неудачная попытка удаления информации об отделе из БД



	1	2	3
1	department	'Deployment and Integration'	[{'Roman','Noyer',32000},{'Suren','Melkonyan',38000}]
2	department	'Testing'	[{'Kate','Torres',35000}]
3	department	'Development'	[{'David','Qwerty',45000},{'Nik','Zxcver',25000}]
4	department	'Business analysis'	[{'Anna','Krug',25000}]
5			
6			
7			
8			

Рисунок 10 – Таблица БД после некоторых изменений

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы я изучил принципы функционального программирования на примере языка Erlang. Я узнал об основных особенностях языка и провел сравнительный анализ с другими языками программирования. Было разработано приложение, ориентированное на работу с системой управления базами данных (СУБД) Mnesia в целях закрепления полученных знаний о языке.

Изучение данной темы оказалось крайне полезным и помогло взглянуть на разработку ПО со стороны функционального программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Cesarini, F., Erlang Programming / F. Cesarini, S.Thompson. — O'REILLY, 2009.

2 Logan, M., Erlang and OTP in Action / M. Logan, E. Merritt, R. Carlsson. — MANNING, 2011.

3 Kessin, Z., Building Web Applications with Erlang / Z. Kessin. — O'REILLY, 2012.

4 Hebert, F., Learn You Some Erlang for Great Good! / F. Hebert. — Ebook, 2013.

5 Erlang Programming Language http://www.erlang.org

6 Erlang/OTP · GitHub https://github.com/erlang

ПРИЛОЖЕНИЕ А
Листинг программы

Структура записи `department` описана в заголовочном файле `company_db.hrl`.

```
-record( department, { name, employees } ).
```

Фунции, работающие напрямую с СУБД Mnesia, описаны в модуле `company_db.erl`.

```
-module( company_db ).

-export( [ createDepartmentDB/0 ] ).
-export( [ addDepartment/2, readDepartment/1, deleteDepartment/1,
                addEmployeeToDepartment/4, deleteEmployeeFromDepartment/4 ] ).

-include_lib("stdlib/include/qlc.hrl").
-include( "company_db.hrl" ).

createDepartmentDB() ->
        mnesia:create_table( department,
                [ { attributes, record_info( fields, department ) } ] ).

addDepartment( Name, Employees ) ->
        { NameFromTable, _ } = readDepartment( Name ),
        case NameFromTable of
                Name -> ok;
                _ ->
                        NewDepartment = #department{
                                name = Name,
                                employees = Employees },
                        mnesia:transaction( fun() ->
                                mnesia:write( NewDepartment )
                                end ),
                        ok.
        end.

readDepartment( Name ) ->
        { atomic, [ { department, NameDepartment, Employees } ] } =
        mnesia:transaction( fun() ->
                  mnesia:read( { department, Name } )
                end ),
```

```
33          { NameDepartment, Employees }.
34
35  deleteDepartment( Name ) ->
36  not_empty.
37          { NameFromTable, Employees } = readDepartment( Name ),
38          case Employees of
39                  [] ->
40                          mnesia:transaction( fun() ->
41                                  mnesia:delete( { department, Name } )
42                          end ),
43                          ok;
44                  [ List ] -> not_empty
45          end.
46
47  deleteFullDepartment( Name ) ->
48          mnesia:transaction( fun() ->
49                  mnesia:delete( { department, Name } )
50          end ).
51
52  addEmployeeToDepartment( NameDepartment, NameEmployee,
53          SurnameEmployee, SalaryEmployee ) ->
54          { NameDep, Employees } = readDepartment( NameDepartment ),
55          NewEmployees = Employees ++
56              [ { NameEmployee, SurnameEmployee, SalaryEmployee } ],
57          deleteFullDepartment ( NameDep ),
58          addDepartment( NameDepartment, NewEmployees ).
59          ok.
60
61  deleteEmployeeFromDepartment( NameDepartment, NameEmployee,
62          SurnameEmployee, SalaryEmployee ) ->
63          { NameDep, Employees } = readDepartment( NameDepartment ),
64          NewEmployees = deleteEmployeeFromList( Employees,
65              { NameEmployee, SurnameEmployee, SalaryEmployee } ),
66          deleteFullDepartment ( NameDep ),
67          addDepartment( NameDepartment, NewEmployees ).
68          ok.
69
70  deleteEmployeeFromList( [], { Employee } ) -> [];
71  deleteEmployeeFromList( [ FirstEmployee, OtherEmployees ], Employee ) ->
72          case FirstEmployee of
73                  Employee ->
```

```
74                          deleteEmployeeFromList( [ OtherEmployees ], Employee );
75                  _ ->
76                          [ FirstEmployee ] ++
77                  deleteEmployeeFromList( [ OtherEmployees ], Employee )
78                  end.
```

Код программы модуля для работы с Mnesia представлен в файле `work_db.erl`.

```
1 -module( work_db ).
2
3 -import( company_db, [ createDepartmentDB/0,
4         addDepartment/2, readDepartment/1, deleteDepartment/1,
5         addEmployeeToDepartment/4, deleteEmployeeFromDepartment/4 ] ).
6 -import( lists, [foreach/2] ).
7
8 -export( [ init/0, start/0,
9         newDepartment/1, printDepartment/1, delDepartment/1,
10        addNewEmployeeToDepartment/4, deleteOldEmployeeFromDepartment/4 ] ).
11
12 -include( "company_db.hrl" ).
13
14 init() ->
15        mnesia:create_schema( [ node() ] ),
16        mnesia:start(),
17        createDepartmentDB(),
18        mnesia:stop().
19
20 start() ->
21        mnesia:start().
22
23 newDepartment( Name ) ->
24        addDepartment( Name, [] ).
25
26 printDepartment( Name ) ->
27        readDepartment( Name ).
28
29 delDepartment( Name ) ->
30        deleteDepartment( Name ).
31
32 addNewEmployeeToDepartment( NameDepartment, NameEmployee,
33            SurnameEmployee, SalaryEmployee ) ->
34        addEmployeeToDepartment( NameDepartment, NameEmployee,
```

```
35                  SurnameEmployee, SalaryEmployee ).
36
37 deleteOldEmployeeFromDepartment( NameDepartment, NameEmployee,
38              SurnameEmployee, SalaryEmployee ) ->
39         deleteEmployeeFromDepartment( NameDepartment, NameEmployee,
40                  SurnameEmployee, SalaryEmployee ).
```