Вычислительные машины и программное обеспечение

УДК 004.432

Г.С. Петросян

ЯЗЫКОВЫЕ СРЕДСТВА ПОДДЕРЖКИ СИСТЕМАТИЧЕСКОЙ ОБРАБОТКИ ОШИБОК

G.S. Petrosyan

LANGUAGE SUPPORT FOR SYSTEMATIC ERROR HANDLING

Введено разделение ошибок времени выполнения на ожидаемые и неожиданные; предложены критерии, которым должны соответствовать языковые средства поддержки систематической обработки ошибок. Проанализированы популярные подходы к обработке ошибок с использованием механизма исключений и кодов возврата. Предложен подход, согласно которому обработка неожиданных ошибок должна состоять в автоматическом поддержании инвариантов состояния программы. На основе проведенного анализа предложен ряд языковых средств, соответствующий современным требованиям к механизму поддержки систематической обработки ошибок.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ОБРАБОТКА ОШИБОК. ИСКЛЮЧЕНИЯ. КОДЫ ВОЗВРАТА. ИНВАРИАНТЫ.

Runtime errors are divided into expected and unexpected; requirements for language support for systematic error handling are proposed. Popular approaches to error handing — exceptions and return codes — are analyzed. Automatic treatment of unexpected errors by means of restoring program state invariants is proposed. Based on the conducted analysis, several mechanisms of language support for systematic error handling which satisfy stated requirements are proposed.

PROGRAMMING LANGUAGES. ERROR HANDLING. EXCEPTIONS. RETURN CODES. INVARIANTS.

Определимся с терминологией: *ошибкой* будем называть ситуацию, при которой часть программы не смогла успешно завершить действие, которое должна была выполнить. Например, в ходе расчета произошло деление на ноль, либо была предпринята попытка чтения из несуществующего файла. Причинами ошибок могут быть как внешние условия (такие, как отсутствие доступа в Интернет или неисправность подключенного оборудования), так и внутренние противоречия (такие, как попытка получить доступ к первому элементу пустого массива) — в данной статье не делается разделения ошибок на какие-либо классы.

Несмотря на то что начинающие программисты часто стремятся игнорировать обработку ошибок, их опытные коллеги знают, что систематическая обработка ошибок не менее важна, чем «основная» функциональность программы [1]. Объем кода, посвященного обработке ошибок, при этом может достигать 60 % [2]. Неудивительно, что многие языки программирования включают специальные средства, целью которых является поддержка систематической обработки ошибок.

Наиболее распространенное средство поддержки систематической обработки ошибок — механизм *исключений* (exceptions). Первым описанием механизма, похожего на используемый сегодня в «промышлен-

ных» языках (C++, Java, C#), можно считать [3]; формализован подход в [4]. Большое количество современных языков (в т. ч. Java, Scala, C#, F#, Python, PHP, Ruby и JavaScript) копируют вариант, впервые появившийся в C++ [5], и используют его как единственный механизм обработки ошибок. Однако, как показывает анализ реальных программ, такой подход не может считаться удовлетворительным [6].

Вместе с тем некоторые достаточно новые языки – Go [7] и Rust [8] – предполагают обработку ошибок, в т. ч. и без использования специальных языковых средств (тем не менее механизмы, похожие на исключения, присутствуют в обоих языках). Такой подход демонстрирует что исключения, в их общепринятом виде, не являются однозначно предпочтительным средством обработки ошибок.

В данной статье рассматриваются общие моменты обработки ошибок; анализируется обработка ошибок с помощью распространенного механизма исключений, а также без помощи специальных языковых средств (с использованием кода возврата). Выделяются сильные и слабые стороны каждого из подходов и предлагается новый, более предпочтительный в соответствии с критериями анализа.

Обработка ошибок

Какой должна быть реакция программы на возникновение ошибки в вызываемой подпрограмме? Для анализа разделим ошибки на ожидаемые и неожиданные. *Ожидаемыми* назовем ошибки, для которых существует предпочтительный механизм обработки, вытекающий из общей логики работы программы. Например, в случае, если веб-браузер не смог загрузить изображение, вместо него на странице должно быть использовано специальное, встроенное в браузер. Все другие ошибки назовем *неожиданными*.

Как следует из определений, специфическая реакция на ожидаемые ошибки – это часть общей логики работы программы. Определим такую реакцию как *альтернативу* – «если не получилось, то ...» в бизнес-логике. Напротив, обработка неожиданных ошибок не связана с бизнес-логикой и заключается в следующем:

освобождении ресурсов, используемых текущей операцией;

опционально: создание записи об ошибке журнале выполнения;

информировании вызывающего контекста (либо пользователя).

Отметим, что ошибки, вызванные нарушением контракта вызывающим контекстом (например, передача аргумента, для которого не выполняется предусловие), являются неожиданными, и сказанное выше относится к ним тоже.

Исходя из проведенного разделения, сформулируем требования к механизму обработки ошибок.

1. Обработка ожидаемых ошибок (использование *альтернативы*) должна выглядеть как составная часть бизнес-логики.

2. Неожиданные ошибки не должны игнорироваться по умолчанию.

3. Код обработки неожиданных ошибок не должен смешиваться с бизнес-логикой.

4. Код обработки неожиданных ошибок должен быть максимально кратким.

5. Информация об ошибке должна быть максимально подробной.

6. Информация об ошибке должна передаваться в вызывающий контекст автоматически.

7. Освобождение ресурсов при обработке ошибки должно происходить автоматически.

Рассмотрим обработку ошибок с помощью механизма исключений и с помощью кодов возврата, в т. ч. и в контексте указанных требований. Предполагается достаточное знакомство с языками C и C++.

Обработка ошибок с помощью механизма исключений. Будем рассматривать широко распространенный вариант механизма исключений, впервые введенный в C++ (большой сравнительный обзор других вариантов можно найти в [9]). Сильными сторонами обработки ошибок с помощью механизма исключений являются:

невозможность игнорировать ошибку, не обработанную явно;

автоматическое информирование вызывающего контекста об ошибке;

возможность передать подробную информацию об ошибке;

упрощение интерфейсов подпрограмм; отсутствие необходимости изменения ин-

терфейса в случае, если для подпрограммы, ранее не предполагающей ошибочного завершения, оно стало возможным («совместимость с будущим», forward compatibility).

Среди слабых сторон можно отметить:

громоздкий синтаксис обработки ожидаемых ошибок;

последовательную семантику исключений (только одно исключение может быть активно в некоторый момент времени; оно требует немедленной обработки), что не является оптимальным для параллельных программ;

более низкую скорость работы – либо в случае успешного, либо в случае ошибочного выполнения, в зависимости от реализации механизма исключений [10];

добавление неявных потоков управления.

Из последнего пункта следует возможность возникновения большого количества «ошибочных», не предусмотренных авторами, состояний программы в результате возникновения исключений и, в частности, несовместимость механизма с обработкой ошибок без использования исключений.

Проанализируем использование механизма исключений в соответствии с приведенными критериями 1–7.

Механизм исключений не делает различий между ожидаемыми и неожиданными ошибками. Как следствие, требование 1 не выполняется – использование альтернативы не выглядит частью бизнес-логики. Требование 3, напротив, выполняется.

Требование 2 выполняется – оно было одной из причин ввода механизма исключений в язык.

Требования 4, 6 и 7 выполняются, если используется (там, где это применимо) подход RAII [5]. Однако там, где он неприменим, освобождение ресурсов необходимо производить вручную, и код становится более громоздким и подверженным ошибкам.

Требование 5 выполняется только в случае, если программист будет самостоятельно перехватывать исключение по мере его продвижения по вызывающим контекстам и дополнять его необходимой информацией. Таким образом, требования 5 и 6 конфликтуют.

Обработка ошибок с помощью кодов возврата. Сильными сторонами обработки ошибок с помощью использования кодов возврата являются:

отсутствие в языке дополнительных механизмов, которыми должен овладеть программист;

простота использования;

предсказуемая скорость выполнения кода;

совместимость с кодом, написанным на других языках (ABI-совместимость).

Среди слабых сторон можно выделить:

отсутствие обработки (игнорирование) ошибок по умолчанию;

громоздкий код обработки ошибок;

невозможность передать подробную информацию об ошибке;

отсутствие «совместимости с будущим» – потенциальная необходимость изменения интерфейса подпрограмм в случае появления новых ошибочных ситуаций;

сложность написания безопасного относительно исключений (exception-safe) кода [11].

Относительно критериев 1–7 можно отметить следующее: выполняется лишь важное требование 1; все прочие не выполняются.

Предлагаемый подход

Необходимость наличия единого механизма обработки ошибок. Несложно увидеть, что механизмы исключений и кодов возврата можно рассматривать как дополняющие друг друга: исключения хорошо работают для неожиданных ошибок, а коды возврата – для ожидаемых; исключения хорошо подходят для последовательного кода, а коды ошибок – для параллельного. Является ли правильным совмещение этих механизмов, подобно тому, как предлагают авторы Go и Rust?

Приведем простое рассуждение, которое показывает, что подобный подход неправилен. Что должен использовать автор подпрограммы при определении ошибочной ситуации? Так как он не может знать, каким образом будет использоваться его подпрограмма (например, будет ли эта ошибка ожидаемой или нет), то он не в состоянии обоснованно выбрать подходящий механизм.

Некоторые придерживаются противоположной точки зрения: автор подпрограммы может знать, например, какая ошибка будет

ожидаемой, а какая — нет. В случае подпрограммы открытия файла для чтения, ошибка «файл не существует» может рассматриваться как ожидаемая. Однако несложно представить пример, в котором это не так: допустим, если это служебный файл, до этого созданный самой программой в ходе выполнения. Очевидно, что в этом случае ошибка будет являться неожиданной.

Таким образом, в языке должен присутствовать единый механизм обработки ошибок, который предоставлял бы вызывающему (а не вызываемому) контексту выбирать семантику обработки.

Информирование об ошибке. Код, обнаруживший ошибку, должен сообщить о ней с использованием `fail`, аналога `throw` в C++ (здесь и далее для примеров используется модельный язык с синтаксисом и семантикой, близкими к C++):

```
double sqrt(double x) {
    if (x < 0) {
        fail InvalidArgumentError(x);
    }
}
```

Семантика fail аналогична throw, за исключением того, что в данные об ошибке автоматически добавляется информация о функции и строчке, в которой она произошла. Информирование вызывающего контекста об ошибке происходит автоматически, аналогично механизму исключений.

Обработка ожидаемой ошибки. Для обработки ожидаемой ошибки используется механизм `inline try`:

```
x = try sqrt(y);
switch x.kind() {
case error:
case result:
}
```

Механизм `inline-try` преобразует выражение, возвращающее данные типа X, в выражение, возвращающее данные типа `X|error` — объединение, наподобие `union` и `boost::variant<>` в терминах C++. Работа с результатом `inline-try` является явной частью бизнес-логики.

Обработка неожиданных ошибок. Желательно, чтобы обработка неожиданных ошибок не затрудняла чтение кода бизнес-логики. Кроме этого, неожиданные ошибки должны обрабатываться в обязательном порядке.

Предлагаемый механизм основан на наблюдении, что в обработке неожиданных ошибок главное — не собственно «обработка» ошибок, а поддержание инвариантов программы при условии возникновения ошибки. На основе этого наблюдения предлагается два механизма — `onerror` и `finally` (см. [12]):

```
x = action();
onerror rollback();
finally cleanup();
```

Код, ассоциированный с `onerror`, выполнится только при возникновении ошибки; при наличии нескольких подобных блоков в контексте их выполнение будет происходить в обратном порядке (аналогично выполнению деструкторов локальных объектов). Код, ассоциированный с `finally`, будет выполняться всегда при выходе из текущего контекста (опять же, аналогично коду деструкторов локальных объектов).

Такой подход позволяет группировать операции восстановления инвариантов и освобождения ресурсов с операциями бизнес-логики. Его преимуществом является хорошая масштабируемость на большое количество операций, а также легкость написания и чтения кода:

```
x = action1();
onerror rollback1();
finally cleanup1();

y = action2();
onerror rollback2();
finally cleanup2();

z = action3();
onerror rollback3();
finally cleanup3();
```

Соответствующий код на Java/C# (простая трансляция):

```
try {
    x = action1();
    try {
        y = action2();
        try {
            z = action3();
        } catch (Exception e) {
```

```
            rollback3();
            throw e;
        } finally {
            cleanup3();
        }
    } catch (Exception e) {
        rollback2();
        throw e;
    } finally {
        cleanup2();
    }
} catch (Exception e) {
    rollback1();
    throw e;
} finally {
    cleanup1();
}
```

Механизм `onerror / finally` похож на механизм деструкторов локальных объектов в C++. В связи с этим предлагается уточнить и расширить понятие деструктора: деструктор заменяется специальными функциями `onerror()`, `onsuccess()` и `onexit()`, отвечающими за поддержание инвариантов в случае ошибки, успешного выполнения, и за освобождение ресурсов, соответственно.

Дополнение информации об ошибке. Нередко вызывающие контексты обладают полезной информацией о неожиданной ошибке. Эта информация, например, может пригодиться при отображении пользователю сообщения об ошибке. В широко используемом механизме исключений, однако, для добавления подобной информации к исключению код должен явно его перехватить, изменить и повторно возбудить, что не соответствует выдвинутому требованию автоматической обработки неожиданных ошибок.

Для решения указанной проблемы предлагается добавить в язык механизм аннотирования текущего действия информацией, потенциально полезной для пользователя:

```
f = fopen(filename, "rb");
do "reading from file", filename
read_from_file(f);
```

В случае возникновения неожиданной ошибки в `read_from_file`, генерируемое исключение будет автоматически дополнено именем файла — информацией, недоступной на момент возникновения ошибки.

Подобный механизм позволит автоматически получать после возникновения ошибки данные, что значительно улучшит информативность сообщения об ошибке для пользователя (по сравнению с существующими подходами, в т. ч. отображением трассировки стека выполнения).

В данной статье проанализировано использование популярных подходов к обработке ошибок — механизма исключений и кодов возврата. Введено разделение ошибок на ожидаемые и неожиданные и предложены критерии, которым должны соответствовать языковые средства поддержки систематической обработки ошибок. Предложена точка зрения, согласно которой обработка неожиданных ошибок должна быть автоматической и заключаться в поддержании инвариантов состояния программы.

На основании проведенного анализа и выдвинутых критериев предложен ряд языковых средств, который, будучи реализованным, позволит:

использовать единый механизм для обработки всех возможных ошибок;

объединить код обработки ожидаемых ошибок с бизнес-логикой (частью которой он должен, по определению, являться);

обеспечить автоматическую обработку всех неожиданных ошибок, в т. ч. освобождение ресурсов и восстановление необходимых инвариантов состояния;

сделать компактным, легким в сопровождении и явно выделенным из бизнес-логики код поддержания инвариантов и освобождения ресурсов;

обеспечить автоматическое уведомление вызывающих контекстов об ошибке и автоматическое дополнение сообщения об ошибке контекстно-зависимой информацией.

Естественно, предложенный подход обладает и недостатками, большинство из которых являются следствием его схожести с механизмом обработки исключений. В связи с этим в дальнейшей работе планируется исследовать вопрос использования модели компиляции, автоматически (либо по запросу программиста) заменяющей использование семантики исключений семантикой кодов возврата.

СПИСОК ЛИТЕРАТУРЫ

1. **Shah, H.** Understanding exception handling: Viewpoints of novices and experts [Электронный ресурс] / H. Shah, C. Gorg, M. Harrold // Software Engineering, IEEE Transactions on. – 2010. – Vol. 36. – № 10. – P. 1–12.

2. **Gehani, N.** Exceptional C or C with Exceptions [Text] / N. Gehani //Software: Practice and Experience. – 1992. – Vol. 22. – № 10. –P. 827–848.

3. **Goodenough, J.B.** Exception handling: issues and a proposed notation [Text] / J.B. Goodenough // Communications of the ACM. – 1975. – Vol. 18. – № 12. – P. 683–696.

4. **Cristian, F.** Exception handling and software fault tolerance [Электронный ресурс] / F. Cristian // Computers, IEEE Transactions on. – 1982. – Vol. 100. – № 6. – P. 531–540.

5. **Страуструп, Б.** Дизайн и эволюция языка C++ [Текст]/ Б. Страуструп. – СПб.: Питер, 2006. – 448 с.

6. **Cabral, B.** Exception handling: A field study in Java and .NET [Text] / B. Cabral, P. Marques // ECOOP 2007. – Springer Berlin Heidelberg, 2007. – P. 151–175.

7. Go Programming Language [Электронный ресурс]/ Режим доступа: http://golang.org (Дата обращения 25.06.2013)

8. Rust Programming Language [Электронный ресурс]/ Режим доступа: http://rust-lang.org (Дата обращения 25.06.2013)

9. **Garcia, A.** A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software [Text] / A. Garcia, C. Rubira, A. Romanovsky, J. Xu // J. of systems and software. – 2001. – Vol. 59. – № 2. – P. 197–222.

10. **Goldthwaite, L.** Technical report on C++ performance [Электронный ресурс] / L. Goldthwaite // ISO/IEC PDTR. – 2006. – Vol. 18015.

11. **Stroustrup, B.** Exception Safety: Concepts and Techniques [Text] / B. Stroustrup // Advances in exception handling techniques. – Springer Berlin Heidelberg, 2001. – P. 60–76.

12. **Alexandrescu, A.** Generic: Change the Way You Write Exception-Safe Code Forever [Text] / A. Alexandrescu, P. Marginean // Dr. Dobb's Journal. – CMP Media LLC., 2003.

REFERENCES

1. **Shah H., Gorg C., Harrold M.** Understanding exception handling: Viewpoints of novices and experts / Software Engineering, IEEE Transactions on. – 2010. – Vol. 36. – № 10. – P. 1–12.

2. **Gehani N.** Exceptional C or C with Exceptions / Software: Practice and Experience. – 1992. – Vol. 22. – № 10. – P. 827–848.

3. **Goodenough J.B.** Exception handling: issues and a proposed notation / Communications of the ACM. – 1975. – Vol. 18. – № 12. – P. 683–696.

4. **Cristian F.** Exception handling and software fault tolerance / Computers, IEEE Transactions on. – 1982. – Vol. 100. – № 6. – P. 531–540.

5. **Straustrup B.** Dizain i evoliutsiia iazyka C++. – St.-Petersburg: Piter, 2006. – 448 s. (rus)

6. **Cabral B., Marques P.** Exception handling: A field study in Java and .NET / ECOOP 2007. – Springer Berlin Heidelberg, 2007. – P. 151–175.

7. Go Programming Language. Availiable at http://golang.org (Accessed 25.06.2013)

8. Rust Programming Language. Availiable at http://rust-lang.org (Accessed 25.06.2013)

9. **Garcia A., Rubira C., Romanovsky A., Xu J.** A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software / Journal of systems and software. – 2001. – Vol. 59. – № 2. – P. 197–222.

10. **Goldthwaite L.** Technical report on C++ performance / ISO/IEC PDTR. – 2006. – Vol. 18015.

11. **Stroustrup B.** Exception Safety: Concepts and Techniques / Advances in exception handling techniques. – Springer Berlin Heidelberg, 2001. – P. 60–76.

12. **Alexandrescu A., Marginean P.** Generic: Change the Way You Write Exception-Safe Code Forever / Dr. Dobb's Journal. – CMP Media LLC., 2003.

ПЕТРОСЯН Григорий Сергеевич — *аспирант кафедры прикладной математики Санкт-Петербургского государственного политехнического университета.*
195251, Россия, Санкт-Петербург, ул. Политехническая, д. 29.
E-mail: gregory.petrosyan@gmail.com

PETROSYAN, Gregory S. *St. Petersburg State Polytechnical University.*
195251, Politekhnicheskaya Str. 29, St.-Petersburg, Russia.
E-mail: gregory.petrosyan@gmail.com