Obsługa błędów

czyli jak sobie radzić z prawem Murphy'ego

Daniel Janus

http://korpus.pl/~nathell/blog

26 czerwca 2008

"Anything that can go wrong, will"

— przypisywane Edwardowi A. Murphy'emu (1918—1990)

1 Błędy i bugi

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego:
jeśli coś się może nie udać, to się nie uda

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego:
jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego:
jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci
- Zapis na dysk się nie powiedzie

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego:
jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci
- Zapis na dysk się nie powiedzie
- Nie uda się wysłać pakietu w sieć

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego:
jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci
- Zapis na dysk się nie powiedzie
- Nie uda się wysłać pakietu w sieć
- Serwer, z którym rozmawiamy, się wykrzaczy

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego: jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci
- Zapis na dysk się nie powiedzie
- Nie uda się wysłać pakietu w sieć
- Serwer, z którym rozmawiamy, się wykrzaczy

Bugi — nieoczekiwane zachowania programu

Co jest błędem, a co bugiem?

Błędy — konsekwencje prawa Murphy'ego: jeśli coś się może nie udać, to się nie uda

- Zabraknie pamięci
- Zapis na dysk się nie powiedzie
- Nie uda się wysłać pakietu w sieć
- Serwer, z którym rozmawiamy, się wykrzaczy

Bugi — nieoczekiwane zachowania programu

[+/-] Exception:

Failure invoking listener method 'public java.lang.String com.sentivision.lstv.mws.admin.pages.UserForm.doSubmit() throws java.lang.Exception' on $UserForm_33@3c1[UserForm]: javax.ejb.EJBException

[+/-] Exception:

javax.ejb.EJBException

Stack Trace:

- com.sun.ejb.containers.BaseContainer.processSystemException(BaseContainer.java:3869)

sentivision

Powiązanie

Błędy niekoniecznie muszą oznaczać bugi, ale nieobsłużenie błędu prawie na pewno spowoduje powstanie buga

Obsługa błędów w C

Język C nie ma żadnego dedykowanego mechanizmu kontroli błędów. Jeśli wywołanie funkcji nie powiedzie się, musi ona zwrócić wartość specjalną oznaczającą powstanie błędu.

Obsługa błędów w C

Język C nie ma żadnego dedykowanego mechanizmu kontroli błędów. Jeśli wywołanie funkcji nie powiedzie się, musi ona zwrócić wartość specjalną oznaczającą powstanie błędu.

Należy **zawsze** sprawdzać tę wartość.

Obsługa błędów w C

Język C nie ma żadnego dedykowanego mechanizmu kontroli błędów. Jeśli wywołanie funkcji nie powiedzie się, musi ona zwrócić wartość specjalną oznaczającą powstanie błędu.

Należy **zawsze** sprawdzać tę wartość.

Przykłady

- `malloc()` — zwraca `NULL`, jeśli nie udało się przydzielić pamięci
- Większość funkcji systemowych zwraca `-1` w przypadku niepowodzenia i ustawia zmienną `errno`

Szóste przykazanie

6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.

— Henry Spencer,
The Ten Commandments for C Programmers (Annotated Edition)

Grzechy przeciw szóstemu przykazaniu

Grzechy lekkie

Kto sprawdza, co zwróciło `printf()` albo `fprintf()`?

Grzechy przeciw szóstemu przykazaniu

Grzechy lekkie

Kto sprawdza, co zwróciło `printf()` albo `fprintf()`?

Jeśli wypisanie tekstu się nie uda, to rzadko jest sens reagować na taką sytuację inaczej niż ignorując błąd. (Wypisywać komunikat o błędzie? A jeśli znowu się nie uda?)

Grzechy przeciw szóstemu przykazaniu

Grzechy lekkie

Kto sprawdza, co zwróciło `printf()` albo `fprintf()`?

Jeśli wypisanie tekstu się nie uda, to rzadko jest sens reagować na taką sytuację inaczej niż ignorując błąd. (Wypisywać komunikat o błędzie? A jeśli znowu się nie uda?)

Ale co w sytuacji, gdy piszemy za pomocą `fprintf()` do pliku opartego via `fdopen()` na deskryptorze gniazda sieciowego?

Grzechy przeciw szóstemu przykazaniu (cd.)

...i cięższe

Często zdarza nam się nie sprawdzać wartości `malloc()`.

Grzechy przeciw szóstemu przykazaniu (cd.)

...i cięższe

Często zdarza nam się nie sprawdzać wartości `malloc()`.

Pod Linuksem to najczęściej nie boli: `malloc()` rzadko zwraca `NULL`, co więcej, nawet jeśli zwróci wartość nienullową, nie ma gwarancji, że pamięć jest rzeczywiście dostępna (optymistyczna strategia przydziału).

Grzechy przeciw szóstemu przykazaniu (cd.)

...i cięższe

Często zdarza nam się nie sprawdzać wartości `malloc()`.

Pod Linuksem to najczęściej nie boli: `malloc()` rzadko zwraca `NULL`, co więcej, nawet jeśli zwróci wartość nienullową, nie ma gwarancji, że pamięć jest rzeczywiście dostępna (optymistyczna strategia przydziału).

...Chyba że parametr VM jądra `overcommit_memory` jest ustawiony na 2.

Jak sobie zaoszczędzić pisania

Wyabstrahować obsługę błędów

Jeśli obsługa pewnego rodzaju błędu (np. braku pamięci) jest taka sama w całym programie, to warto umieścić ją w osobnej funkcji.

Wady mechanizmu z C

Wady mechanizmu z C

- Konieczność żmudnego sprawdzania kodu błędu za każdym razem

Wady mechanizmu z C

- Konieczność żmudnego sprawdzania kodu błędu za każdym razem
- Kod każdej nietrywialnej funkcji jest usiany sprawdzeniami

Wady mechanizmu z C

- Konieczność żmudnego sprawdzania kodu błędu za każdym razem
- Kod każdej nietrywialnej funkcji jest usiany sprawdzeniami
- Brak możliwości oddzielenia obsługi błędów od właściwego kodu

Wady mechanizmu z C

- Konieczność żmudnego sprawdzania kodu błędu za każdym razem
- Kod każdej nietrywialnej funkcji jest usiany sprawdzeniami
- Brak możliwości oddzielenia obsługi błędów od właściwego kodu

Szczególnie dobrze widać to na przykładzie kodu przydzielającego i używającego kilku zasobów.

Wady mechanizmu z C (cd.)

```
int funkcja() {
   int retval = -1;
   resource resource1 = allocate_resource();
   if (resource1 == NULL)
      goto err1;
   resource resource2 = allocate_resource();
   if (resource2 == NULL)
      goto err2;
   resource resource3 = allocate_resource();
   if (resource3 == NULL)
      goto err3;
   if (do_something(resource1, resource2, resource3) == 0)
      retval = 0;
   deallocate_resource(resource3);
err3:
   deallocate_resource(resource2);
err2:
   deallocate_resource(resource1);
err1:
   return retval;
}
```

Wyjątki

Wyjątki

- Mechanizm pozwalający na oddzielenie obsługi błędów od miejsca ich wystąpienia

Wyjątki

- Mechanizm pozwalający na oddzielenie obsługi błędów od miejsca ich wystąpienia
- Wprowadzony po raz pierwszy w PL/I ok. 1970

Wyjątki

- Mechanizm pozwalający na oddzielenie obsługi błędów od miejsca ich wystąpienia
- Wprowadzony po raz pierwszy w PL/I ok. 1970
- ```
  try {
      actual_code();
  } catch (exception e) {
      handle_exception(e);
  } catch (another_exception e) {
      handle_another_exception(e);
  }
  ```
  ```

Wyjątki

- Mechanizm pozwalający na oddzielenie obsługi błędów od miejsca ich wystąpienia
- Wprowadzony po raz pierwszy w PL/I ok. 1970
- ```
  try {
      actual_code();
  } catch (exception e) {
      handle_exception(e);
  } catch (another_exception e) {
      handle_another_exception(e);
  }
  ```

  W ciele funkcji `actual_code()` albo wołanych przez nią funkcyj może pojawić się operacja zwana *rzuceniem* albo *zgłoszeniem* wyjątku
  ```

Wyjątki

- Mechanizm pozwalający na oddzielenie obsługi błędów od miejsca ich wystąpienia
- Wprowadzony po raz pierwszy w PL/I ok. 1970
- ```
  try {
      actual_code();
  } catch (exception e) {
      handle_exception(e);
  } catch (another_exception e) {
      handle_another_exception(e);
  }
  ```

  W ciele funkcji `actual_code()` albo wołanych przez nią funkcyj może pojawić się operacja zwana *rzuceniem* albo *zgłoszeniem* wyjątku
- Powoduje ona zwinięcie stosu do najbliższego odpowiadającego bloku `catch`
  ```

Wyjątki w różnych językach

Rodzaje wyjątków

Wyjątki w różnych językach

Rodzaje wyjątków

- W językach obiektowych można definiować klasy wyjątków

Wyjątki w różnych językach

Rodzaje wyjątków

- W językach obiektowych można definiować klasy wyjątków
- Jeśli dany blok `try/catch` nie obsługuje wyjątku danej klasy, jest ona pomijana i następuje poszukiwanie handlera wyżej, aż do wierzchołka stosu

Wyjątki w różnych językach

Rodzaje wyjątków

- W językach obiektowych można definiować klasy wyjątków
- Jeśli dany blok `try/catch` nie obsługuje wyjątku danej klasy, jest ona pomijana i następuje poszukiwanie handlera wyżej, aż do wierzchołka stosu
- W Javie wszystkie wyjątki wyprowadzone są z klasy `Throwable`, a własne należy wyprowadzać z `Exception` lub jej podklas

Wyjątki w różnych językach

Rodzaje wyjątków

- W językach obiektowych można definiować klasy wyjątków
- Jeśli dany blok `try/catch` nie obsługuje wyjątku danej klasy, jest ona pomijana i następuje poszukiwanie handlera wyżej, aż do wierzchołka stosu
- W Javie wszystkie wyjątki wyprowadzone są z klasy `Throwable`, a własne należy wyprowadzać z `Exception` lub jej podklas
- W C++ rzucane mogą być dowolne obiekty (ale zaleca się wyprowadzanie wyjątków z `std::exception`)

Wyjątki w różnych językach (cd.)

Klauzula `finally`

Wyjątki w różnych językach (cd.)

Klauzula `finally`

- Może się pojawić w bloku `try` po klauzulach `catch`

Wyjątki w różnych językach (cd.)

Klauzula `finally`

- Może się pojawić w bloku `try` po klauzulach `catch`
- Kod objęty tą klauzulą wykonywany jest niezależnie od tego, czy ciało bloku `try` wykona się normalnie czy zostanie rzucony wyjątek

Wyjątki w różnych językach (cd.)

Klauzula `finally`

- Może się pojawić w bloku `try` po klauzulach `catch`
- Kod objęty tą klauzulą wykonywany jest niezależnie od tego, czy ciało bloku `try` wykona się normalnie czy zostanie rzucony wyjątek
- Najczęściej używana do zapewnienia zwalniania zasobów

Wyjątki w różnych językach (cd.)

Klauzula `finally`

- Może się pojawić w bloku `try` po klauzulach `catch`
- Kod objęty tą klauzulą wykonywany jest niezależnie od tego, czy ciało bloku `try` wykona się normalnie czy zostanie rzucony wyjątek
- Najczęściej używana do zapewnienia zwalniania zasobów
- Dostępna w Javie, C#, Pythonie, Common Lispie (pod nazwą `UNWIND-PROTECT`)
  ale nie w OCamlu (choć można ją łatwo dodać za pomocą OCamlowych możliwości rozszerzeń składniowych)
  ale nie w C++ (który woli używać techniki RAII do zwalniania zasobów)

Resource Acquisition Is Initialization

The C++ way

Pozyskanie zasobu jest inicjalizacją

Resource Acquisition Is Initialization

The C++ way

Pozyskanie zasobu jest inicjalizacją

- Wzorzec projektowy postulujący, że pozyskanie zasobu powinno być powiązane z konstrukcją obiektu, a jego zwolnienie — z automatyczną destrukcją

Resource Acquisition Is Initialization

The C++ way

Pozyskanie zasobu jest inicjalizacją

- Wzorzec projektowy postulujący, że pozyskanie zasobu powinno być powiązane z konstrukcją obiektu, a jego zwolnienie — z automatyczną destrukcją
- W C++ pamięć traktuje się pod tym względem tak jak każdy inny zasób

Resource Acquisition Is Initialization

The C++ way

Pozyskanie zasobu jest inicjalizacją

- Wzorzec projektowy postulujący, że pozyskanie zasobu powinno być powiązane z konstrukcją obiektu, a jego zwolnienie — z automatyczną destrukcją
- W C++ pamięć traktuje się pod tym względem tak jak każdy inny zasób
- Kłóci się to z odśmiecaniem — w językach odśmiecanych programista rzadko troszczy się o czas życia obiektu

Resource Acquisition Is Initialization — cd.

```
class resource {
  private:
    resource_t res;
  public:
    resource() { res = allocate_resource(); }
    ~resource() { deallocate_resource(res); }
};

void funkcja() {
  try {
    resource res1, res2; // initialize resources
    do_something(res1, res2);
  } catch (...) {
    std::cerr << "Error!" << std::endl;
  }
}
```

Wersja z **try/finally**

```
void funkcja() {
  try {
    resource res1 = allocate_resource();
    try {
      resource res2 = allocate_resource();
      do_something(res1, res2);
    } finally {
      deallocate_resource(res2);
    }
  } finally {
    deallocate_resource(res1);
  }
}
```

Wersja z **try/finally**

```
void funkcja() {
  try {
    resource res1 = allocate_resource();
    try {
      resource res2 = allocate_resource();
      do_something(res1, res2);
    } finally {
      deallocate_resource(res2);
    }
  } finally {
    deallocate_resource(res1);
  }
}
```

W niektórych językach (Lisp, Python >= 2.5) można definiować abstrakcje składniowe, pozwalające na napisanie tego samego zwięźlej:

```
(with-resource res1
  (with-resource res2
    (do-something res1 res2)))
```

Wyjątki deklarowane

W Javie lista wyjątków rzucanych przez metodę jest częścią sygnatury metody. Jeśli metoda próbuje rzucać inne wyjątki niż to deklaruje, to skutkuje to błędem kompilacji.

Wyjątki deklarowane

W Javie lista wyjątków rzucanych przez metodę jest częścią sygnatury metody. Jeśli metoda próbuje rzucać inne wyjątki niż to deklaruje, to skutkuje to błędem kompilacji.
Mechanizm ten nie stosuje się do wyjątków, których nadklasami są `Error` lub `RuntimeException`.

Zalety

Wyjątki deklarowane

W Javie lista wyjątków rzucanych przez metodę jest częścią sygnatury metody. Jeśli metoda próbuje rzucać inne wyjątki niż to deklaruje, to skutkuje to błędem kompilacji.
Mechanizm ten nie stosuje się do wyjątków, których nadklasami są `Error` lub `RuntimeException`.

Zalety

- Możliwe sytuacje błędne są częścią kontraktu funkcji

Wyjątki deklarowane

W Javie lista wyjątków rzucanych przez metodę jest częścią sygnatury metody. Jeśli metoda próbuje rzucać inne wyjątki niż to deklaruje, to skutkuje to błędem kompilacji.
Mechanizm ten nie stosuje się do wyjątków, których nadklasami są `Error` lub `RuntimeException`.

Zalety

- Możliwe sytuacje błędne są częścią kontraktu funkcji
- Znaczne ograniczenie w czasie kompilacji liczby wyjątków docierających do wierzchołka stosu

Wady

Wyjątki deklarowane

W Javie lista wyjątków rzucanych przez metodę jest częścią sygnatury metody. Jeśli metoda próbuje rzucać inne wyjątki niż to deklaruje, to skutkuje to błędem kompilacji.
Mechanizm ten nie stosuje się do wyjątków, których nadklasami są `Error` lub `RuntimeException`.

Zalety

- Możliwe sytuacje błędne są częścią kontraktu funkcji
- Znaczne ograniczenie w czasie kompilacji liczby wyjątków docierających do wierzchołka stosu

Wady

- Zwiększona dyscyplina kontroli wyjątków zachęca do jej obchodzenia nieprzemyślanymi blokami `catch`

Wady wyjątków

Wady wyjątków

- Jeśli wywołanie funkcji zakończy się rzuceniem wyjątku (odpowiadającego błędowi), to trudno naprawić to, co się stało, bez znajomości „bebechów” tej funkcji

Wady wyjątków

- Jeśli wywołanie funkcji zakończy się rzuceniem wyjątku (odpowiadającego błędowi), to trudno naprawić to, co się stało, bez znajomości „bebechów” tej funkcji
- Ale funkcje powinny być „czarnymi skrzynkami” — nie powinno nas obchodzić, jak wywoływana przez nas funkcja wykonuje swoje zadanie

Wady wyjątków

- Jeśli wywołanie funkcji zakończy się rzuceniem wyjątku (odpowiadającego błędowi), to trudno naprawić to, co się stało, bez znajomości „bebechów” tej funkcji
- Ale funkcje powinny być „czarnymi skrzynkami” — nie powinno nas obchodzić, jak wywoływana przez nas funkcja wykonuje swoje zadanie
- Sytuacja jeszcze się komplikuje, jeśli dany błąd może teoretycznie być obsłużony na wiele możliwych sposobów

Wady wyjątków

- Jeśli wywołanie funkcji zakończy się rzuceniem wyjątku (odpowiadającego błędowi), to trudno naprawić to, co się stało, bez znajomości „bebechów” tej funkcji
- Ale funkcje powinny być „czarnymi skrzynkami” — nie powinno nas obchodzić, jak wywoływana przez nas funkcja wykonuje swoje zadanie
- Sytuacja jeszcze się komplikuje, jeśli dany błąd może teoretycznie być obsłużony na wiele możliwych sposobów
- W szczególności, jeśli np. skończyło się miejsce na dysku, to zamiast wyświetlenia komunikatu o błędzie może chcemy wywalić śmieci z `/tmp` i spróbować jeszcze raz?

Poza wyjątkami: sytuacje i restarty

Przychodzi pracownik do szefa i mówi: jest taka sytuacja...

Poza wyjątkami: sytuacje i restarty

Przychodzi pracownik do szefa i mówi: jest taka sytuacja...

- Wyobraźmy sobie, że mamy funkcję `Szef()`, która wywołuje funkcję `Pracownik()`.

Poza wyjątkami: sytuacje i restarty

Przychodzi pracownik do szefa i mówi: jest taka sytuacja...

- Wyobraźmy sobie, że mamy funkcję `Szef()`, która wywołuje funkcję `Pracownik()`.
- W modelu wyjątków: jeśli ta druga funkcja rzuci wyjątek `NieZdazeException`, to stos jest natychmiast zwijany i blok `catch` w funkcji `Szef()` musi ogarnąć sytuację.

Poza wyjątkami: sytuacje i restarty

Przychodzi pracownik do szefa i mówi: jest taka sytuacja...

- Wyobraźmy sobie, że mamy funkcję `Szef()`, która wywołuje funkcję `Pracownik()`.
- W modelu wyjątków: jeśli ta druga funkcja rzuci wyjątek `NieZdazeException`, to stos jest natychmiast zwijany i blok `catch` w funkcji `Szef()` musi ogarnąć sytuację.
- Ale może funkcja `Pracownik()` byłaby w stanie coś zrobić, gdyby `Szef()` powiedział, jaką strategię wyjścia należy obrać (np. na podstawie priorytetu zadania)?

Poza wyjątkami: sytuacje i restarty

Przychodzi pracownik do szefa i mówi: jest taka sytuacja...

- Wyobraźmy sobie, że mamy funkcję `Szef()`, która wywołuje funkcję `Pracownik()`.
- W modelu wyjątków: jeśli ta druga funkcja rzuci wyjątek `NieZdazeException`, to stos jest natychmiast zwijany i blok `catch` w funkcji `Szef()` musi ogarnąć sytuację.
- Ale może funkcja `Pracownik()` byłaby w stanie coś zrobić, gdyby `Szef()` powiedział, jaką strategię wyjścia należy obrać (np. na podstawie priorytetu zadania)?
- Na tym opiera się mechanizm sytuacji (*conditions*): oddzielamy politykę (**co** zrobić?) od obsługi (**jak** to zrobić?)

Pseudokod: szef

```
void Szef() {
  while (task = GetNextTask()) {
    try {
      Pracownik(task, task.dueDate());
    } catch (NieZdaze cond) {
      if (case.priority == critical)
        restart cond, zostanPoGodzinach;
      else
        restart cond, fajrant;
    }
  }
}
```

Pseudokod: pracownik

```
void Pracownik(Task task, DateTime dueDate) {
  try {
    do {
      perform(task);
      if (dueDate - currentDate() <
            complexity * (1 - progress(task)))
        throw NieZdaze;
    } while (more());
  } restart (zostanPoGodzinach) {
    do {
      bluzg();
      perform(task);
    } while (more());
  } restart (fajrant) {
  }
}
```

Dziękuję za uwagę!

sentivision