Zarządzanie zasobami. Wzorce projektowe

Autorzy: Michael Kircher, Prashant Jain
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 83-246-0102-3
Tytuł oryginału: Pattern-Oriented Software Architecture, Patterns for Resource Management
Format: B5, stron: 352

Techniki implementacji wydajnych mechanizmów zarządzania zasobami

- Pozyskiwanie zasobów
- Wykorzystywanie zasobów
- Zwalnianie zasobów

Efektywne zarządzanie zasobami ma kluczowe znaczenie dla funkcjonowania oprogramowania. Niezależnie od tego, czy są to małe systemy instalowane w urządzeniach przenośnych, czy rozbudowane aplikacje korporacyjne, musimy mieć pewność, że pamięć, wątki, pliki i połączenia sieciowe są zarządzane w sposób, który zapewnia właściwe i wydajne działanie systemu. Konieczność stosowania efektywnych metod zarządzania zasobami zbyt często jest odkrywana w późnych fazach projektów informatycznych. Wprowadzanie zmian jest wtedy trudne i kosztowne.

Książka „Zarządzanie zasobami. Wzorce projektowe" przedstawia metody implementacji efektywnych mechanizmów zarządzania zasobami w systemach informatycznych. Wzorce przydzielono do trzech grup odpowiadających naturalnemu cyklowi życia zasobów. Każdy wzorzec został zilustrowany przykładem.
Książka zawiera również dwa studia przypadków, które opisują możliwości stosowania przedstawionych wzorców w sieciach komputerowych.

- Przegląd technik zarządzania zasobami
- Stosowanie wzorców projektowych
- Wzorce pozyskiwania zasobów
- Wzorce zarządzania zasobami
- Wzorce zwalniania zasobów

Dzięki zawartym w tej książce wiadomościom stworzysz wydajniejsze oprogramowanie.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

3 Cykl życia zasobów

Nie szukaj duszo nieśmiertelności,
ciesz się raczej tymi zasobami,
które są w twoim zasięgu.

Pindar

Kiedy już pozyskamy niezbędny zasób, musimy znaleźć sposób na efektywne zarządzanie jego cyklem życia. Zarządzanie zasobami wiąże się oczywiście z ich udostępnianiem użytkownikom, obsługą zależności międzyzasobowych, pozyskiwaniem — w razie konieczności — wszelkich zasobów zależnych oraz zwalnianiem zasobów, kiedy okaże się, że nie są one już potrzebne.

Wzorzec projektowy Caching (zob. strona 121) opisuje sposób zarządzania cyklem życia często wykorzystywanych zasobów, który pozwala znacznie ograniczyć koszty ich ponownego pozyskiwania i zwalniania, zachowując jednocześnie podstawowe właściwości (identyfikator) tych zasobów. Wzorzec Caching jest niezwykle popularny — powszechnie stosuje się go między innymi w wysoce skalowalnych rozwiązaniach korporacyjnych. Wzorzec

projektowy Pooling (zob. strona 138) — podobnie jak wzorzec Caching — optymalizuje procesy pozyskiwania i zwalniania zasobów, ale — w przeciwieństwie do tamtego wzorca — nie zachowuje unikatowych identyfikatorów tych zasobów. Wzorzec Pooling jest więc dobrym rozwiązaniem w przypadku zasobów bezstanowych, które wymagają stosunkowo niewielu działań w fazie inicjalizacji lub nie wymagają ich wcale. Podobnie jak Caching, wzorzec projektowy Pooling jest bardzo popularny — można bez trudu wskazać przykłady puli komponentów w architekturach komponentowych lub puli wątków w aplikacjach rozproszonych. Wzorce Caching i Pooling mogą być stosowane wyłącznie dla zasobów wielokrotnego użytku. Oba wzorce stosuje się dla zasobów wielokrotnego użytku z wyłącznym dostępem, które są kolejno wykorzystywane przez wielu użytkowników. Warto jednak pamiętać, że w niektórych przypadkach zastosowanie wzorca Caching lub Pooling także dla współbieżnie wykorzystywanych zasobów wielokrotnego użytku znajduje uzasadnienie. W takim przypadku ani wzorzec Caching, ani wzorzec Pooling w ogóle nie musi „wiedzieć", że pojedyncze zasoby są udostępniane wielu użytkownikom jednocześnie (uczestniczą w przetwarzaniu współbieżnym), ponieważ wszelkie operacje i tak dotyczą tylko zasobów „pobranych" z pamięci podręcznej lub puli.

Dwa lub wiele składników programu, czyli np. pozyskanych zasobów, użytkowników zasobów lub dostawców zasobów, może ze sobą współpracować i wprowadzać odpowiednie zmiany do danego systemu informatycznego. W tego typu sytuacjach mówi się, że wspomniane składniki są aktywne i zdolne do uczestnictwa w działaniach skutkujących zmianami. W takim przypadku bardzo ważne jest utrzymywanie spójnego stanu systemu mimo zmian generowanych przez aktywnych uczestników przetwarzania. Wzorzec projektowy Coordinator (zob. strona 155) daje nam pewność, że realizacja zadań wymagających udziału wielu uczestników nie spowoduje utraty spójności i jako takie nie obniżą one ogólnej stabilności systemu.

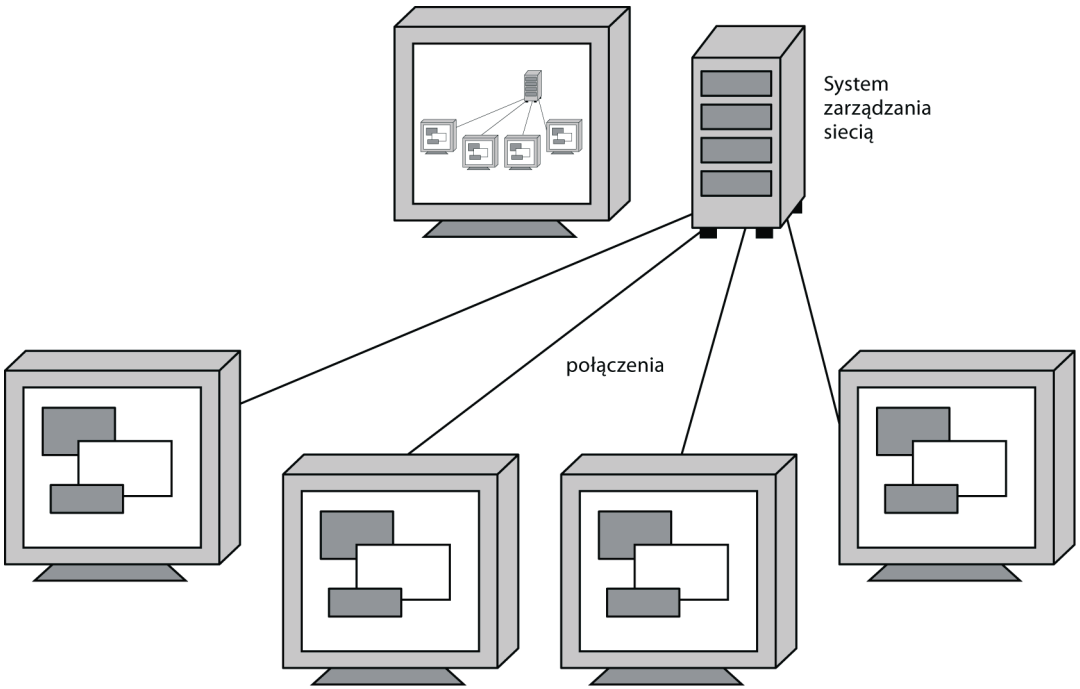
Wzorzec Resource Lifecycle Manager (zob. strona 175) zarządza wszystkimi zasobami danego systemu informatycznego, co oznacza, że zwalnia z obowiązku właściwego zarządzania cyklem życia zasobów zarówno same zasoby, jak i ich użytkowników. Wzorzec projektowy Resource Lifecycle Manager odpowiada za zarządzania cyklem życia wszystkich typów zasobów, włącznie z zasobami wielokrotnego użytku i jednorazowego użytku.

Caching

Wzorzec **Caching** opisuje sposób unikania kosztownych operacji ponownego pozyskiwania zasobów, ponieważ nie zwalnia zasobów zaraz po ich użyciu. Zasoby zachowują swoją identyfikatory i są składowane w pamięci zapewniającej możliwie szybki dostęp. Aby uniknąć konieczności ponownego pozyskiwania zasobów, wzorzec projektowy Caching przewiduje możliwość ich ponownego wykorzystywania.

Przykład Wyobraź sobie system zarządzający siecią, który musi monitorować stan wielu elementów sieciowych. Tego rodzaju systemy zwykle implementuje się w architekturze trójwarstwowej. Użytkownicy końcowi mogą korzystać z takiego systemu za pośrednictwem warstwy prezentacji, która zwykle ma postać graficznego interfejsu użytkownika (GUI). Warstwa środkowa (pośrednicząca), która powinna zawierać całą logikę biznesową, współpracuje zarówno z warstwą utrwalania, jak i z fizycznymi elementami sieciowymi. Ponieważ typowa sieć może się składać z tysięcy takich elementów, ustanawianie trwałych połączeń pomiędzy warstwą środkową, serwerem aplikacji i wszystkimi tymi elementami byłoby bardzo kosztowne. Z drugiej strony, użytkownik końcowy może (za pośrednictwem odpowiednich funkcji interfejsu GUI) wskazać dowolne elementy sieciowe, aby uzyskać szczegółowe informacje na temat tych elementów. System zarządzania siecią musi odpowiadać na żądania użytkowników w skończonym czasie, zatem powinien gwarantować krótkie czasy oczekiwania pomiędzy żądaniem użytkownika (wskazaniem konkretnego elementu sieciowego) a odpowiedzią systemu (wizualizacją właściwości tego elementu).

Z ustanawianiem nowych połączeń sieciowych dla wszystkich wybieranych przez użytkownika elementów sieciowych oraz niszczenie tych połączeń już po wykorzystaniu wiązałoby się z ogromnym obciążeniem procesora w ramach serwera aplikacji. Co więcej, średni czas dostępu do elementu sieciowego byłby bardzo długi.



Kontekst Systemy, które wielokrotnie uzyskują dostęp do tego samego zbioru zasobów i które wymagają odpowiedniej optymalizacji w obszarze wydajności.

Problem Wielokrotne powtarzanie operacji pozyskiwania, inicjalizacji i zwalniania tego samego zasobu powoduje zupełnie niepotrzebne opóźnienia. Jeśli jeden lub wiele komponentów systemu uzyskuje dostęp do tego samego zasobu i — tym samym — wielokrotnie powtarza operacje pozyskiwania i inicjalizacji, koszt wyrażany w cyklach procesora i ogólnej wydajności systemu może być bardzo wysoki. Poprawa wydajności wymaga więc znacznego obniżenia kosztów pozyskiwania, dostępu i zwalniania najczęściej wykorzystywanych zasobów.

Aby skutecznie rozwiązać ten problemu, należy uwzględnić następujące **czynniki**:

- **Wydajność.** Należy zminimalizować koszty wielokrotnie powtarzanych operacji pozyskiwania, inicjalizacji i zwalniania zasobów.

- **Złożoność.** Nasze rozwiązanie z pewnością nie powinno niepotrzebnie komplikować operacji pozyskiwania i zwalniania zasobów. Co więcej, stosowane rozwiązanie nie powinno dodawać niepotrzebnego poziomu pośredniczenia w dostępie do zasobów.
- **Dostępność.** Rozwiązanie problemu powinno zapewniać dostępność zasobów także wtedy, gdy dostawcy zasobów są tymczasowo niedostępni.
- **Skalowalność.** Stosowane rozwiązanie powinno być skalowalne przede wszystkim w wymiarze liczby zasobów.

Rozwiązanie

Należy w sposób przezroczysty składować zasoby w buforze zapewniającym szybki dostęp, nazywanym pamięcią podręczną. Oznacza to, że kiedy użytkownik zażąda ponownego dostępu, pobieramy dany zasób z pamięci podręcznej, zamiast ponownie pozyskiwać ten zasób za pośrednictwem dostawcy (np. zarządzającego zasobami systemu operacyjnego). Zasoby w pamięci podręcznej są identyfikowane według unikatowych właściwości (jakichś cech charakterystycznych), a więc wskaźnika, referencji lub klucza głównego.

Zachowując często wykorzystywane zasoby i nie zwalniając ich po każdym użyciu, możemy w prosty sposób uniknąć kosztów związanych z ponownymi pozyskiwaniem i wielokrotnym zwalnianiem. Stosowanie pamięci podręcznej ułatwia też zarządzanie komponentami, które uzyskują dostęp do takich zasobów.

Kiedy zasoby w pamięci podręcznej nie będą już potrzebne, należy je zwolnić. Od implementacji samej pamięci podręcznej zależy, jak i kiedy usuwać niepotrzebne zasoby. Odpowiednie zachowania można kontrolować za pomocą starannie dobieranych strategii.

Struktura

Na poniższej liście wymieniono i krótko opisano uczestników wzorca projektowego Caching:

- **Użytkownik** zasobów wykorzystuje dany zasób.
- **Zasób** jest jakimś bytem, np. połączeniem.
- **Pamięć podręczna zasobów** buforuje zasoby zwalniane przez ich użytkowników.
- **Dostawca zasobów** posiada i zarządza wieloma zasobami.

Przedstawione poniżej karty CRC ilustrują sposób, w jaki współpracują ze sobą uczestnicy tego wzorca.

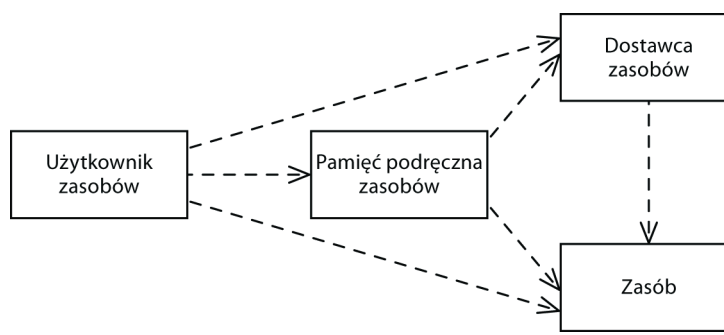
Class **Użytkownik zasobów**	*Współpracownicy*
Zakres odpowiedzialności • Jako pierwszy pozyskuje zasób od jego dostawcy • Używa zasobu • Zwalnia i zwraca zasób do pamięci podręcznej zasobów • Pozyskuje zasoby z pamięci podręcznej zasobów	• Zasób • Dostawca zasobów • Pamięć podręczna zasobów

Class **Zasób**	*Współpracownicy*
Zakres odpowiedzialności • Jest pozyskiwany od dostawcy zasobów i wykorzystywany przez użytkownika zasobów	

Class **Pamięć podręczna zasobów**	*Współpracownicy*
Zakres odpowiedzialności • Buforuje zasoby • Ostatecznie usuwa zasoby	• Zasób • Dostawca zasobów

Class **Dostawca zasobów**	*Współpracownicy*
Zakres odpowiedzialności • Posiada wiele zasobów i zarządza nimi	• Zasób

Strukturę wzorca projektowego Caching przedstawiono na poniższym diagramie klas.

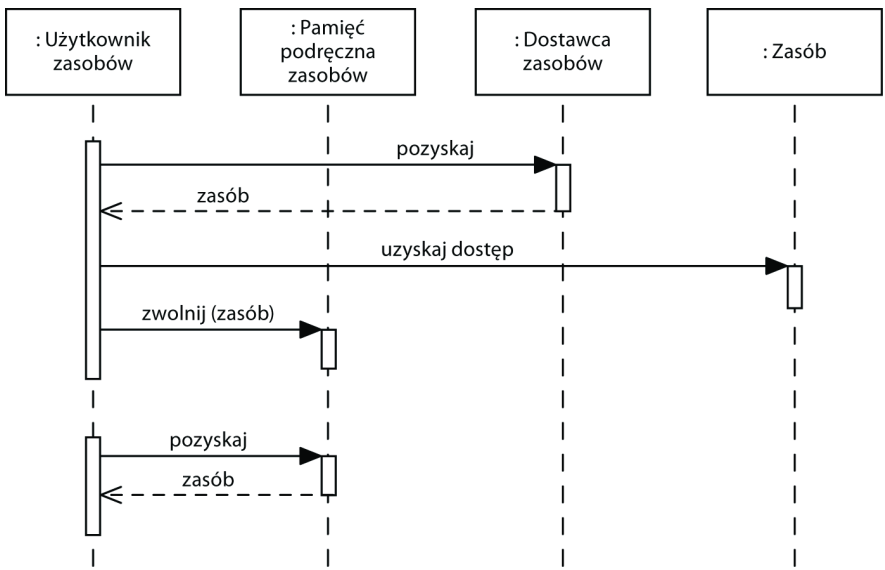


Rola dostawcy zasobów, np. systemu operacyjnego, sprowadza się w takim przypadku do zarządzania zasobami do momentu ich pierwszego pozyskania przez użytkownika. Użytkownik uzyskuje następnie dostęp do pozyskanych w ten sposób zasobów. Kiedy zasoby nie będą już potrzebne, użytkownik zwraca je do pamięci podręcznej (bufora). Użytkownik wykorzystuje tę pamięć do pozyskiwania tych zasobów, których ponownie potrzebuje. Pozyskiwanie zasobów z pamięci podręcznej jest tańsze (przynajmniej od

pozyskiwania bezpośrednio od dostawców) zarówno w wymiarze obciążenia procesora, jak i opóźnień czasowych.

Dynamika Na poniższym rysunku przedstawiono sposób pozyskiwania zasobu przez użytkownika bezpośrednio od jego dostawcy. Użytkownik uzyskuje następnie dostęp do tego zasobu. Użyty w ten sposób zasób jest zwracany do pamięci (zamiast do swojego dostawcy).

Kiedy okaże się, że użytkownik ponownie musi uzyskać dostęp do tego samego zasobu, nie korzysta już z dostawcy, tylko od razu pobiera ten zasób z pamięci podręcznej.



Implementacja Aby zaimplementować wzorzec projektowy Caching, należy wykonać następujące kroki:

1. **Dobór zasobów.** Należy wybrać zasoby, które rzeczywiście skorzystają na stosowaniu techniki wykorzystującej pamięć podręczną. W większości przypadków będą to zasoby, których pozyskiwanie jest kosztowne i które są wykorzystywane stosunkowo często. Wzorzec projektowy Caching jest bardzo często wprowadzany jako technika optymalizacji oprogramowania po zidentyfikowaniu wąskich gardeł obniżających jego łączną wydajność.

   W systemach rozproszonych pamięć podręczna może występować w dwóch postaciach: pamięci klienta i pamięci serwera. Pamięć podręczna po stronie klienta daje pewne oszczędności w wymiarze

obciążenia połączeń sieciowych i — tym samym — czasu potrzebnego do wielokrotnego przesyłania danych pomiędzy serwerem a klientem. Z drugiej strony, pamięć podręczna po stronie serwera jest skutecznym rozwiązaniem w sytuacji, gdy liczne żądania wielu klientów dotyczą pozyskiwania i zwalniania tego samego zasobu.

2. **Określenie interfejsu pamięci podręcznej.** Skoro użytkownik ma zwalniać i ponownie pozyskiwać zasoby bezpośrednio z pamięci podręcznej, należy zaprojektować odpowiedni interfejs dla tej pamięci. Taki interfejs powinien oczywiście udostępniać metody `release()` i `acquire()` (reprezentujące odpowiednio operacje zwalniania i pozyskiwania zasobów).

```
➥ public interface Cache {
    public void release (Resource resource);
    public Resource acquire (Identity id) throws ResourceNotFound;
  }
```

Konstruując powyższy interfejs zakładaliśmy, że istnieje i jest dostępny zasób z unikatowym identyfikatorem, co nie we wszystkich przypadkach musi mieć miejsce. Może się zdarzyć, że identyfikator zasobu będzie wymagał wyznaczenia takiego identyfikatora na podstawie swoich właściwości (tożsamości)[1]. ❑

Metoda `release()` jest wywoływana przez użytkownika zasobu w momencie, w którym zdecyduje o zwolnieniu zasobu — zasób jest wówczas zwracany do pamięci podręcznej zamiast do swojego oryginalnego dostawcy.

3. **Implementacja pamięci podręcznej.** Właściwa implementacja metod `acquire()` i `release()` interfejsu wzorca projektowego Caching jest kluczem do funkcjonalności pamięci podręcznej.

➥ Poniższy fragment kodu zawiera implementację metody `release()`, która jest wywoływana w chwili zwalniania danego zasobu przez jego użytkownika:

```
public class CacheImpl implements Cache {
  public void release (Resource resource) {
    String id = resource.getId ();
    map.put (id, resource);
  }
  //…

  private HashMap map = new HashMap ();
}
```
❑

[1] Nie będziemy tutaj szczegółowo omawiali tego zagadnieniem, ponieważ wykracza ono poza zakres tematyczny niniejszej książki.

Metoda `release()` dodaje zasób do odpowiedniej struktury danych (w tym przypadku typu `HashMap`), dzięki czemu będzie można później ten zasób pozyskać, przekazując jego identyfikator. Zastosowano strukturę `HashMap` ze względów optymalizacyjnych, ponieważ czas wyszukiwania w tablicy (mapie) asocjacyjnej jest stały. Wzorzec projektowy Comparand [Costanza, 2001] wprowadza pewne koncepcje w zakresie porównywania identyfikatorów. W zależności od rodzaju zasobu, może się okazać, że w pierwszej kolejności należy wyznaczyć jego identyfikator. W tym przypadku zasób zawiera już odpowiedni identyfikator.

Metoda `acquire()` naszej implementacji pamięci podręcznej powinna odpowiadać za odnajdywanie zasobu w strukturze tablicy asocjacyjnej według przekazanego na wejściu identyfikatora. Jeśli operacja pozyskania zasobu z pamięci podręcznej zakończy się niepowodzeniem, co oznacza, że nie udało się znaleźć zasobu z danym identyfikatorem, pamięć podręczna teoretycznie może podjąć próbę pozyskania tego zasobu bezpośrednio od jego dostawcy. Więcej informacji na ten temat znajdziesz w omówieniu wersji „Przezroczysta pamięć podręczna" w punkcie „Wersje".

Poniższy fragment kodu zawiera przykładową implementację metody `acquire()`.

```
public class CacheImpl implements Cache {
  public Resource acquire (Identity id) throws ResourceNotFound
{
    Resource resource = (Resource)map.get (id);
    if (resource == null)
      throw new ResourceNotFound ("Zasób z id" + id.toString ()
+ " nie znaleziony!");
    return resource;
  }
}
```

❑

4. **Określenie sposobu integracji z pamięcią podręczną** (krok opcjonalny). Jeśli chcemy zintegrować pamięć podręczną z naszym systemem w sposób przezroczysty, powinniśmy rozważyć użycie wzorca projektowego Interceptor [Schmidt, 2000] lub Cache Proxy [Buschmann, 1996]. Wprowadzenie któregoś z tych wzorców pozwoli ograniczyć złożoność operacji jawnego zwalniania i ponownego pozyskiwania zasobów z pamięci podręcznej, ponieważ zapewni przezroczystość odpowiednich działań. Więcej informacji temat sposobów zapewniania przezroczystości operacji na zasobach składowanych w pamięci podręcznej znajdziesz

w omówieniu wersji „Przezroczysta pamięć podręczna" w punkcie „Wersje". Warto jednak pamiętać, że opisane sposoby nie eliminują dodatkowego poziomu pośrednictwa związanego z koniecznością przeszukiwania pamięci podręcznej.

5. **Wybór strategii usuwania zasobów z pamięci podręcznej.** Zasoby składowane w pamięci podręcznej wymagają dodatkowej przestrzeni pamięciowej. Jeśli tego rodzaju zasoby nie są wykorzystywane przez długi czas, ich dalsze składowanie staje się bezcelowe i obniża łączną efektywność systemu. Należy wówczas użyć wzorca projektowego Evictor (zob. strona 221), który będzie stopniowo usuwał z bufora niepotrzebne zasoby. Istnieje wiele sposobów integrowania wzorca Evictor ze wzorcem Caching. Przykładowo mechanizmy wzorca Evictor można wywoływać albo w ciele metody `release()`, albo automatycznie, w regularnych odstępach czasu. Tego rodzaju działania mają oczywiście wpływ na łączną przewidywalność systemu. Co więcej, istnieje wiele różnych sposobów konfiguracji wzorca projektowego Evictor, który może stosować rozmaite strategie wyboru zasobów przeznaczonych do usunięcia, np. począwszy od najdłużej nieużywanych (ang. *Least Recently Used — LRU*), począwszy od najrzadziej używanych (ang. *Least Frequently Used — LFU*) lub innych strategii właściwych dla danej dziedziny. Można się oczywiście posłużyć wzorcem Strategii [Gamma, 1995].

6. **Zapewnianie spójności.** Wiele zasobów ma przypisane stany, które należy odpowiednio inicjalizować podczas tworzenia zasobów. Co więcej, kiedy na którymś z tego rodzaju zasobów wykonujemy operację zapisu, musimy zapewnić spójność pomiędzy oryginalnym zasobem zarządzanym przez swojego dostawcę a jego kopią składowaną w pamięci podręcznej. Ewentualna zmiana oryginalnego zasobu wymaga wykonania wywołań zwrotnych, które zaktualizują jej kopię w pamięci podręcznej. Jeśli natomiast zmieni się sama kopia, większość implementacji pamięci podręcznej stosuje strategię propagowania operacji zapisu, zgodnie z którą zmiany zastosowane na kopii zasobu są automatycznie wprowadzane zarówno w tej kopii, jak i w oryginalnym zasobie. Do implementacji tego typu funkcjonalności zwykle wykorzystuje się mechanizm Synchronizer (obiektu synchronizującego). Oznacza to, że w analizowanym scenariuszu Synchronizer jest istotnym składnikiem wzorca projektowego Caching. Niektóre implementacje pamięci podręcznej dodatkowo optymalizują swoją funkcjonalność,

wprowadzając złożoną logikę utrzymywania spójności pomiędzy oryginalnymi zasobami a ich kopiami.

Do podejmowania decyzji, kiedy należy synchronizować oryginalny zasób z jego kopią, można użyć wzorca projektowego Strategy [Gamma, 1995]. W niektórych przypadkach tylko specjalne działania, np. operacje zapisu, będą wymagały natychmiastowej synchronizacji; w pozostałych przypadkach wystarczy synchronizować zasoby w regularnych odstępach czasu. Synchronizacja może też być wywoływana przez takie zdarzenia zewnętrzne jak aktualizacje oryginalnych zasobów przez innych użytkowników.

➥ Jeśli w analizowanym przykładzie systemu zarządzania siecią fizyczne dane o elemencie sieciowym ulegną zmianie, także reprezentacja tego elementu przechowywana w pamięci podręcznej musi zostać odpowiednio zmodyfikowana. Podobnie jeśli użytkownik zmieni ustawienia jakiegoś elementu sieciowego, wprowadzone przez niego zmiany należy uwzględnić na poziomie odpowiedniego urządzenia. ❑

Przykładowe rozwiązanie

Wyobraźmy sobie system zarządzania siecią, którego zadaniem jest monitorowanie stanu wielu elementów sieciowych. Warstwa środkowa tego systemu wykorzystuje wzorzec projektowy Caching, za pomocą którego zaimplementowano pamięć podręczną połączeń z tymi elementami sieciowymi. W odpowiedzi na podjętą przez użytkownika próbę uzyskania dostępu do konkretnego elementu sieciowego system pozyskuje połączenie z tym elementem. Kiedy okaże się, że takie połączenie nie jest już potrzebne, następuje jego dodanie do pamięci podręcznej. Jeśli w przyszłości pojawi się kolejne żądanie takiego połączenia, zostanie ono pozyskane z tej pamięci, dzięki czemu unikniemy dużo większych kosztów jego pozyskiwania od oryginalnego dostawcy.

Kolejne połączenia z pozostałymi elementami sieciowymi będą ustanawiane w odpowiedzi na generowane przez użytkowników próby pierwszego dostępu. Kiedy kontekst użytkownika przełączy się na inny element sieciowy, połączenie zostanie zwrócone do pamięci (bufora) połączeń. Jeśli jakiś użytkownik zażąda dostępu do tego samego elementu sieciowego, istniejące połączenie zostanie wykorzystane ponownie. Z dostępem do pozyskanego wcześniej połączenia (występującego w roli zasobu wielokrotnego użytku) nie będą się wiązały żadne dodatkowe opóźnienia.

Wersje

Transparent Cache (przezroczysta pamięć podręczna). Jeśli pamięć podręczna musi zostać zintegrowana z systemem w sposób przezroczysty, do pozyskiwania zasobów żądanych przez klienta powinniśmy użyć wzorca projektowego Lazy Acquisition (zob. strona 70). Takie rozwiązanie jest możliwe tylko wtedy, gdy pamięć podręczna „wie", jak pozyskiwać i inicjalizować takie zasoby. Leniwe pozyskiwanie zasobów pozwala na całkowite uniezależnienie użytkownika zasobów od funkcjonowania pamięci podręcznej.

Read-Ahead Cache (pamięć podręczna wypełnianą z wyprzedzeniem). W sytuacji gdy zasoby są pozyskiwane przez wielokrotne stosowanie wzorca projektowego Partial Acquisition (zob. strona 103), efektywność systemu można znacznie podnieść stosując pamięć podręczną wypełnianą z wyprzedzeniem. Taka pamięć może pozyskiwać zasoby, zanim dotrą one do systemu właściwe żądania użycia i — tym samym — zapewnić niemal natychmiastową dostępność tych zasobów.

Cached Pool (pula pamięci podręcznej). Stosowanie kombinacji pamięci podręcznej i puli może być skutecznym sposobem budowy wyrafinowanych rozwiązań zarządzających zasobami. Kiedy zaistnieje konieczność usunięcia zasobu z pamięci podręcznej, zamiast zwracać ten zasób do jego dostawcy, można go umieścić w specjalnej puli. Pamięć podręczna pełni więc funkcję swoistego pośrednika w dostępie do zasobów. Zwykle definiuje się dla takiej pamięci podręcznej limit czasowy — jeśli założony czas się wyczerpie, zasób traci swój identyfikator i wraca do puli. Zaletą tego rozwiązania jest niewielka optymalizacja: jeśli zasób, który nie został jeszcze zwrócony do puli, jest przedmiotem żądania, unikamy kosztów dodatkowej inicjalizacji (co jest podstawową zaletą koncepcji pamięci podręcznej).

Layered Cache (wielowarstwowa pamięć podręczna). W skomplikowanych systemach wzorzec projektowy Caching często jest stosowany na wielu poziomach jednocześnie. Przykładowo serwer WebSphere Application Server (WAS) [IBM (a), 2004] stosuje pamięć podręczną w wielu aspektach swojego funkcjonowania. Serwer WAS oferuje możliwość aktywnego przechowywania w pamięci podręcznej wyników metod komponentów EJB. Składowanie poleceń w pamięci podręcznej z myślą o ich ponownym wykorzystywaniu przez kolejne obiekty wywołujące umożliwia obsługę żądań w warstwie logiki biznesowej zamiast w warstwie danych, gdzie

przetwarzanie jest z reguły bardziej kosztowne. WebSphere Application Server oferuje też pamięć podręczną dla tzw. przygotowanych, gotowych wyrażeń, które można konfigurować za pomocą obsługiwanego przez wykorzystywaną bazę danych mechanizmu przetwarzania dynamicznych lub statycznych wyrażeń języka SQL. W zależności od stosowanych wzorców dostępu do danych aplikacji, gotowe wyrażenia serwera WAS mogą się przyczynić do znacznej poprawy wydajności aplikacji. Aby poprawić wydajność operacji JNDI, serwer aplikacji WebSphere stosuje pamięć podręczną redukującą liczbę odwołań do serwera nazw podczas operacji wyszukiwania. I wreszcie serwer WAS oferuje komponenty dostępu do danych, które składują w swojej pamięci podręcznej wyniki zapytań wykonanych na bazie danych.

Skutki stosowania

Z uwagi na dodatkowy poziom pośredniczenia, techniki wykorzystujące pamięć podręczną mogą być źródłem pewnych opóźnień w przetwarzaniu, jednak ogólny bilans wydajności jest pozytywny, ponieważ zasoby są pozyskiwane dużo szybciej.

Stosowanie wzorca projektowego Caching niesie ze sobą wiele **korzyści**:

- **Wydajność.** Szybki dostęp do często wykorzystywanych zasobów jest niewątpliwie największą zaletą stosowania pamięci podręcznej. W przeciwieństwie do wzorca projektowego Pooling (zob. strona 138), pamięć podręczna gwarantuje zachowanie przez zasoby ich tożsamości (unikatowych identyfikatorów). Oznacza to, że w razie konieczności uzyskania ponownego dostępu do tego samego zasobu, nie musimy go poszukiwać poza pamięcią podręczną — wystarczy odnaleźć odpowiednią pozycję w strukturze pamięci podręcznej.

- **Skalowalność.** Jedną z zalet stosowania wzorca projektowego Caching jest brak konieczności każdorazowego pozyskiwania i zwalniania zasobów. Pamięć podręczna z natury rzeczy jest implementowana w taki sposób, pozwalający zachowywać najczęściej wykorzystywane zasoby. Oznacza to, że podobnie jak wzorzec Pooling, także wzorzec Caching redukuje eliminuje koszty związane z pozyskiwaniem i zwalnianiem zasobów. Korzyści wynikające z takiego podejścia są tym większe, im częściej wykorzystujemy dany zasób, zatem wzorzec projektowy Caching w istotny sposób poprawia skalowalność systemu.

- **Złożoność korzystania z zasobów.** Stosując pamięć podręczną możemy mieć pewność, że z perspektywy użytkownika zasobów złożoność operacji ich pozyskiwania i zwalniania nie wzrośnie (mimo dodatkowego kroku mającego na celu sprawdzenie dostępności odpowiednich zasobów w pamięci podręcznej).

- **Dostępność.** Składowanie zasobów w pamięci podręcznej zwiększa ich dostępność w sytuacji, gdy oryginalni dostawcy są czasowo niedostępni — niezależnie od stanu dostawców, zasoby w pamięci podręcznej pozostają dostępne dla swoich użytkowników.

- **Stabilność.** Ponieważ wzorzec projektowy Caching ogranicza liczbę operacji zwalniania i ponownego pozyskiwania zasobów, minimalizuje ryzyko fragmentacji pamięci i — tym samym — prowadzi do większej stabilności systemu. Podobny efekt można osiągnąć stosując wzorzec projektowy Pooling.

Ze stosowaniem wzorca Caching wiążą się także pewne **utrudnienia**:

- **Złożoność synchronizacji.** W zależności od rodzaju zasobu, złożoność stosowanej implementacji może rosnąć z uwagi na konieczność zachowania spójności stanu zasobu w pamięci podręcznej i oryginalnych danych, które są przez ten zasób reprezentowane. Zapewnienie takiej zgodności jest jeszcze trudniejsze w środowiskach podzielonych na klastry (w skrajnych przypadkach tego typu komplikacje mogą w ogóle przekreślić sens stosowania wzorca projektowego Caching).

- **Trwałość.** W razie awarii ewentualne zmiany dokonane na zasobach składowanych w pamięci podręcznej mogą zostać utracone. Można tego problemu uniknąć stosując tzw. synchronizowaną pamięć podręczną.

- **Zajętość pamięci.** Stosowanie pamięci podręcznej zwiększa potrzeby systemu w zakresie zajmowanego obszaru pamięci, ponieważ istnieje ryzyko składowania w pamięci podręcznej niewykorzystanych zasobów. Jeśli jednak zdecydujemy się użyć wzorca projektowego Evictor (zob. strona 221), najprawdopodobniej uda nam się zminimalizować liczbę zasobów niepotrzebnie przechowywanych w pamięci podręcznej.

Mechanizm pamięci podręcznej nie jest najlepszym rozwiązaniem w przypadku aplikacji wymagających stałej dostępności danych dla zasobów, których przetwarzanie obejmuje szczególnie kosztowne operacje. Przykładowo aplikacje sterowane przerwaniami, które na każdym kroku wykorzystują operacje wejścia-wyjścia, oraz tzw. systemy wbudowane (osadzone) bardzo rzadko korzystają ze sprzętowych implementacji pamięci podręcznej.

Ogólna uwaga odnośnie do optymalizacji: Wzorzec projektowy Caching powinien być stosowany szczególnie ostrożnie, jeśli inne próby udoskonalenia, np. optymalizacja operacji pozyskiwania samego zasobu, okażą się nieskuteczne. Pamięć podręczna często jest źródłem dodatkowej złożoności na poziomie implementacji — komplikuje konserwację całego rozwiązania i zwiększa łączny poziom wykorzystania zasobów (np. pamięci), ponieważ zasoby składowane w pamięci podręcznej nie są zwalniane natychmiast po użyciu. Oznacza to, że przed podjęciem decyzji o zastosowaniu tego wzorca należy dokładnie rozważyć wady i zalety w takich aspektach jak wydajność, wykorzystanie zasobów i złożoność systemu.

Znane zastosowania

Sprzętowa pamięć podręczna [Smith, 1982]. Niemal wszystkie procesory (ang. *Central Processing Unit — CPU*) wykorzystują własną, sprzętową pamięć podręczną. Taka pamięć nie tylko skraca średni czas dostępu do danych niezbędnych do przetwarzania, ale też pomaga ograniczyć obciążenie magistrali. Z tych dwóch powodów pamięć podręczna procesora zwykle jest szybsza od pamięci operacyjnej (RAM).

Pamięć podręczna w systemach plików i rozproszonych systemach plików [Nelson, 1988] [Smith, 1985]. Rozproszone systemy plików wykorzystują wzorzec projektowy Caching po stronie serwera, aby ograniczyć liczbę stale powtarzanych operacji odczytu bloków dyskowych. Najczęściej wykorzystywane pliki są przechowywane w pamięci serwera, co znacznie skraca czas dostępu. Systemy plików wykorzystują także pamięć podręczną z blokami plików po stronie klienta — w ten sposób można zredukować obciążenie połączeń sieciowych i ewentualne opóźnienia związane z pobieraniem danych z serwera.

Data Transfer Object [Fowler, 2002]. Technologie oprogramowania pośredniczącego, np. CORBA [Object Management Group (a), 2004] oraz Java RMI [Sun Microsystems (g), 2004], oferują możliwość zdalnego przesyłania całych obiektów (zamiast tradycyjnych technik zdalnego wywoływania metod udostępnianych przez zdalne obiekty). Zdalny obiekt jest przesyłany w całości (przez wartość) pomiędzy klientem a serwerem, dzięki czemu możliwe jest lokalne wywoływane jego metod. W ten sposób da się zminimalizować liczbę zdalnych wywołań metod, które zwykle są dość kosztowne. Obiekt jest lokalnie przechowywany w pamięci podręcznej i reprezentuje właściwy obiekt zdalny. Chociaż opisane rozwiązanie w wielu przypadkach poprawia wydajność, należy pamiętać o konieczności zaimplementowania przez użytkownika odpowiednich mechanizmów synchronizacji lokalnej kopii i oryginalnego obiektu zdalnego.

Serwer proxy dla WWW [Squid Web Proxy Cache, 2004]. Serwer proxy dla WWW (ang. *Web proxy*) jest po prostu serwerem proxy pracującym pomiędzy przeglądarką internetową a serwerami WWW. Żądanie dostępu dociera do tego serwera proxy od przeglądarki internetowej za każdym razem, gdy z któregoś z niezliczonych serwerów WWW należy pobrać i otworzyć jakąś stronę internetową. Serwer proxy zapisuje pobraną przez przeglądarkę z serwera WWW stronę internetową w swojej pamięci podręcznej i zwraca ją w odpowiedzi na wszystkie kolejne żądania tej samej strony. Oznacza to, że serwer proxy dla WWW ogranicza liczbę żądań dostępu do stron internetowych przesyłanych przez internet, ponieważ przechowuje najczęściej żądane strony w lokalnej pamięci podręcznej.

Przeglądarki internetowe. Większość popularnych przeglądarek internetowych, włącznie z takimi produktami jak Netscape [Netscape Browser, 2004] czy Internet Explorer [Microsoft Internet Explorer, 2004], składuje we własnej pamięci podręcznej najczęściej otwierane strony WWW. Jeśli użytkownik ponownie zażąda dostępu do tej samej strony, przeglądarka wczyta jej zawartość z pamięci podręcznej i — tym samym — uniknie kosztownego i czasochłonnego pobierania odpowie strony z witryny internetowej. Do określania optymalnego czasu przechowywania stron w pamięci podręcznej (i właściwego momentu ich usuwania) wykorzystuje się mechanizm znaczników czasowych.

Stronicowanie [Tanenbaum, 2001] [Noble, 2000]. Współczesne systemy operacyjne utrzymują w pamięci tzw. strony, które w wielu sytuacjach pozwalają unikać kosztownych operacji odczytu z dyskowego pliku wymiany. Można przyjąć, że strony przechowywane w pamięci znajdują się swoistej pamięci podręcznej. Dopiero kiedy nie uda się znaleźć jakiejś strony w pamięci tej podręcznej, system operacyjny odwołuje się do dużo wolniejszego, dyskowego pliku wymiany.

Plikowa pamięć podręczna [Kistler, 1992]. Plikowe pamięci podręczne poprawiają wydajność i dostępność, ponieważ umożliwiają lokalną pracę na plikach i katalogach pochodzących z zamontowanych napędów sieciowych (także wtedy, gdy z jakiegoś powodu połączenie z tymi napędami nie będzie możliwe). Pliki i katalogi są kopiowane do pamięci podręcznej i synchronizowane z zawartością oryginalnych dysków sieciowych w momencie nawiązania połączenia z siecią. Najnowsze wersje systemów operacyjnych firmy Microsoft obsługują plikową pamięć podręczną w oparciu o technologię nazwaną *Plikami offline*.

.NET [Richter]. Zbiory danych technologii .NET można traktować jak składowane w pamięci bazy danych. Egzemplarze tych zbiorów są tworzone lokalnie, a za ich wypełnianie danymi odpowiadają zapytania języka SQL przetwarzającej jedną lub wiele tabel bazy danych (za pomocą klasy `SqlDataAdapter`). Od tego momentu aplikacje klienckie mogą uzyskiwać dostęp do tych danych za pomocą technik obiektowych. Ewentualne zmiany zostaną uwzględnione w oryginalnej bazie danych dopiero wtedy, gdy użyjemy wprost klasy `SqlDataAdapter` do zaktualizowania odpowiednich tabel. Spójność reprezentacji obiektowej i oryginalnego źródła danych nie jest zapewniana za pomocą żadnych automatycznych mechanizmów.

Enterprise JavaBeans (EJB) [Sun Microsystems (b), 2004]. Komponenty encyjne (ang. *Entity Beans*) technologii EJB reprezentują w warstwie środkowej (na poziomie serwera aplikacji) informacje składowane w bazie danych. W ten sposób można uniknąć kosztownego wyszukiwania danych (pozyskiwania zasobów) w bazie danych.

Obiektowa pamięć podręczna [Oracle, 2003] [ShiftOne Object Cache, 2004]. Obiektowa pamięć podręczna jest próbą stosowania wzorca projektowego Caching w warunkach przetwarzania

obiektowego. W takim przypadku w roli zasobów występują obiekty, które mają przypisane określone koszty tworzenia i inicjalizacji. Obiektowa pamięć podręczna może wyeliminować przynajmniej część tych kosztownych operacji (pod warunkiem, że sposób korzystania z tych zasobów umożliwia ich składowanie w takim dodatkowym buforze).

Pamięć podręczna danych [Robinson, 1990] [Newport, 2004]. Pamięć podręczna danych jest implementacją wzorca projektowego Caching stosowaną dla danych. Dane są traktowane jak zasób, który w pewnych sytuacjach jest trudny do pozyskania. Takie dane mogą na przykład zawierać skomplikowane i kosztowne obliczenia lub odwołania do innego, zewnętrznego źródła danych. Wzorzec Caching w tej wersji umożliwia wielokrotne wykorzystywanie raz wczytanych danych i — tym samym — eliminuje konieczność kosztownego pozyskiwania danych, kiedy okaże się, że są ponownie potrzebne.

iMerge [iMerge, 2003]. iMerge EMS jest systemem zarządzania elementami wchodzącymi w skład systemu sprzętowego iMerge VoIP (Voice over Internet Protocol), który w roli swojego interfejsu komunikacyjnego wykorzystuje protokół SNMP. System iMerge EMS wykorzystuje wzorzec projektowy Caching do optymalizacji komunikacji pomiędzy elementami sieciowymi.

Zobacz także

Chociaż wzorzec projektowy Pooling (zob. strona 138) pod wieloma względami przypomina wzorzec Caching, pomiędzy nimi istnieje też jedna bardzo istotna różnica. Podstawowym założeniem wzorca Pooling jest wielokrotne wykorzystywanie „anonimowych" zasobów, czyli takich, które nie mają przypisanych unikatowych identyfikatorów. Takie rozwiązanie pomaga uniknąć kosztów ponownego pozyskiwania i zwalniania zasobów. Większość zasobów jest pozbawiona jakichkolwiek cech wyróżniających (identyfikujących) spośród pozostałych zasobów tego samego typu. W przeciwieństwie do wzorca Pooling, we wzorcu Caching każdy zasób składowany w pamięci podręcznej ma przypisany unikatowy identyfikator. Oznacza to, że wzorzec Pooling umożliwia przezroczyste pozyskiwanie zasobów, natomiast we wzorcu Caching za pozyskiwanie zasobów odpowiada ich użytkownik.

Wzorzec projektowy Eager Acquisition (zob. strona 88) zwykle wykorzystuje wzorzec Caching do zarządzania chciwie pozyskiwanymi zasobami.

Możemy użyć wzorca Evictor (zob. strona 221) do usuwania niepotrzebnych danych z pamięci podręcznej.

Wzorzec Resource Lifecycle Manager (zob. strona 175) może wewnętrznie wykorzystywać wzorzec projektowy Caching do zapewniania jak najszybszego dostępu do stanowych zasobów.

Wzorzec Cache Proxy [Buschmann, 1996] może posłużyć do ukrycia efektów ubocznych stosowania pamięci podręcznej. Takie rozwiązanie szczególnie dobrze sprawdza się w implementacjach wzorca Smart Proxy [Hohpe, 2003] [Schmidt, 1998] [Object Computing Interactive, 2004], które przechwytują zdalne wywołania.

Wzorzec projektowy Cache Management (zarządzania pamięcią podręczną) [Grand, 1998] koncentruje się na sposobie łączenia pamięci podręcznej ze wzorcem Manager (menadżera) [Sommerlad, 1998], który centralizuje operacje dostępu, tworzenia i niszczenia obiektów. Definicja wymienionych wzorców została opracowana przede wszystkim z myślą o obiektach i połączeniach z bazami danych w kontekście aplikacji Javy.

Wzorzec Page Cache (pamięć podręczna stron internetowych) [Trowbridge, 2003] jest wyspecjalizowaną wersją wzorca uniwersalnego Caching, w której skrócono czas odpowiedzi na żądania dostępu do dynamicznie generowanych stron internetowych. Pamięć podręczną stron internetowych wykorzystuje się po stronie serwera WWW — serwer umieszcza w niej strony indeksowane według adresów URL. Kiedy serwer WWW ponownie otrzyma żądanie dostępu do strony znajdującej się pod tym samym adresem URL, wykona odpowiednie zapytanie na swojej pamięci podręcznej i zwróci zapisaną tam stronę (zamiast ponownie generować jej dynamiczną zawartość).

Podziękowania Dziękujemy Ralphowi Cabrera za to, że zechciał się z nami podzielić swoim doświadczeniem w zakresie stosowania wzorca projektowego Caching, oraz za to, że wskazał nam konkretny przykład praktycznego stosowania tego wzorca, system iMerge. Chcielibyśmy także podziękować Pascalowi Costanza, naszemu opiekunowi podczas konferencji EuroPLoP 2003, za jego bezcenne komentarze na temat naszej pracy. Jesteśmy także wdzięczni uczestnikom warsztatów pisarskich: Frankowi Buschmannowi, Kevlinowi Henneyowi, Wolfgangowi Herznerowi, Klausowi Marquartowi, Allanowi O'Callaghanowi oraz Markusowi Völterowi.