9 Concurrency Patterns

Das Konzept der nebenläufigen Programmierung kann am Anfang ungewohnt sein, da die Anwendung ein Umdenken zur sequentiellen Programmierung erfordert. Dies bedarf vielleicht zuerst ein wenig Übung. Wenn wir dann einmal die ersten nebenläufigen Programme erstellt haben, werden wir feststellen, dass sich alles sehr schnell wiederholt. Denn eigentlich geht es immer darum, Daten über Channels zu schicken und diese dann innerhalb von Goroutinen zu verarbeiten oder mit Goroutinen andere Goroutinen zu steuern. Wenn wir Nachrichten aus Channels bekommen, steht immer ein Ereignis dahinter, auf das wir reagieren müssen. Dafür gibt es eine Handvoll Patterns, die uns das Leben am Anfang sehr erleichtern. Die wichtigsten Patterns sollen in diesem Kapitel vorgestellt werden.

9.1 Checkliste zu Goroutinen

- Was müssen wir beachten, wenn wir eine Goroutine starten?

Bevor wir mit konkreten Patterns tiefer in die Konstruktion von Goroutinen einsteigen, schauen wir uns ein paar Grundsätze an. Diese Grundsätze sollen uns am Anfang die Arbeit erleichtern.

Bei der Konstruktion von Goroutinen sollten wir uns immer folgende Fragen stellen:

- Wie lange soll die Goroutine laufen?
- Wie wird sie beendet?
- Welche Elemente blockieren den Programmfluss?
- Kann es zu einer Leaking Goroutine kommen?
- Was passiert auf der anderen Seite der Channels?

Gehen wir die Punkte noch einmal kurz durch. Die Frage nach der Laufzeit und dem Beenden der Goroutine muss immer beachtet werden. Hier ist das Context-Pattern (Kapitel 9.4) unser wichtigstes Werkzeug. Denn bei Services, die lange und stabil laufen müssen, ist es wichtig, einmal gestartete Goroutinen gezielt beenden zu können. Wir müssen da-

bei aber aufpassen, dass die Goroutinen auch wirklich beendet werden. Deshalb sollten wir beim Start der Goroutine auch prüfen, dass keine Elemente ungewollt blockieren. Denn blockierende Elemente können zu Leaking Goroutines führen. Allgemein müssen wir bei der Verwendung von Channels auch immer die andere Seite im Hinterkopf behalten.

Leaking Goroutines

Leaking Goroutines sind Goroutinen, die während der Laufzeit nie mehr beendet werden. Bei Programmen, die nur kurz laufen, ist dies nicht so schlimm, da mit dem Beenden des Hauptprogramms auch alle Goroutinen beendet werden. Bei Servern sieht dies jedoch ganz anders aus. Hier kann die Laufzeit mehrere Monate betragen, bis der Server neu gestartet wird. Wenn pro Request eine Goroutine als Leiche im Arbeitsspeicher vorgehalten werden muss, geht früher oder später der Arbeitsspeicher aus.

Für die Prüfung unserer Channels gibt es ein paar Merksätze, die die Funktion von Channels zusammenfassen:

- Daten über einen nil-Channel zu senden, führt zu einem Deadlock.
- Daten aus einen nil-Channel lesen zu wollen, führt zu einem Deadlock.
- Einen nil-Channel zu schließen, führt zu einer Panik.
- Daten über einen geschlossenen Channel zu senden, führt zu einer Panik.
- Daten aus einem geschlossenen Channel zu lesen, liefert den Nullwert des Typs zurück.
- Einen bereits geschlossenen Channel zu schließen, führt zu einer Panik.

Mit diesen Merksätzen können wir nun loslegen, um die einzelnen nebenläufigen Patterns kennenzulernen.

9.2 Goroutinen melden, wenn sie fertig sind

- Wie kann eine Goroutine mitteilen, dass sie mit ihrer Arbeit fertig ist?
- Warum ist das wichtig?

Wenn wir eine Goroutine starten, wird diese aus dem aktuellen Programmfluss herausgenommen und blockiert nicht das Hauptprogramm. Damit wir jedoch mitbekommen, wann eine Goroutine fertig ist, muss diese kommunizieren, wann sie fertig ist. In dem Fall können wir Goroutine und Hauptprogramm wieder synchronisieren.

Wenn eine Goroutine ein Ergebnis ermittelt, wird dieses am Ende der Goroutine über einen Channel gesendet. Da Channels blockieren, werden die Goroutinen auf Sender- und Empfängerseite synchronisiert. Entweder der Empfänger oder der Sender wartet, bis das Ergebnis übermittelt werden kann. Um einer Leaking Goroutine vorzubeugen, sollte der `result`-Channel einen Puffer der Größe 1 besitzen. Denn sollte aus irgendwelchen Gründen kein Empfänger für das Ergebnis mehr zur Verfügung stehen, würde die Goroutine nie beendet werden.

Listing 9–1 *Ergebnis an einen Channel*

```
result := make(chan resultType, 1)
// ...
go func() {
    r := doSomething()
    result <- r
}()
// ...
// Verwende das Ergebnis
ergebnis := <-result
```

Goroutinen können aber auch aktiv melden, dass sie fertig sind. Die einfachste Lösung hierfür ist die Verwendung der `sync.WaitGroup`.

Listing 9–2 *Verwendung der sync.WaitGroup*

```
// WaitGroup
var wg &sync.WaitGroup
go func() {
    doSomething()
    wg.Done()
}()
wg.Wait()
```

Die Verwendung der WaitGroup ist komfortabel, schnell und sicher. Als Variablenname hat sich `wg` durchgesetzt. Wenn möglich, sollten wir diesen Namen also ausschließlich für WaitGroups verwenden.

Wenn wir keine WaitGroup verwenden möchten, bietet sich auch ein Channel an. Diesen schließen wir, sobald die Goroutine fertig ist.

Listing 9–3
Synchronisierung mit einem done-Channel

```
// Channel schließen
done := make(chan struct{})
// ...
go func() {
    doSomething()
    close(done)
}()
// Programmfluss blockiert hier, bis
// done geschlossen wird.
<-done
```

Anstatt den Channel zu schließen, können wir auch eine Nachricht schicken. Dabei sollten wir wieder einen Puffer setzen, denn so sind wir unabhängig von einem Empfänger. Die Goroutine wird in jedem Fall beendet. Der einzige Wermutstropfen ist, dass keine Synchronisierung stattfindet.

Listing 9–4
Erledigt über einen buffered Channel

```
done := make(struct{}, 1)
go func() {
    doSomething()
    done <- struct{}{}
}()
```

Alle der hier vorgestellten Pattern ermöglichen es einer Goroutine, zu kommunizieren, dass sie fertig ist.

9.3 Beenden von Goroutinen

- Wie lassen sich Goroutinen so aussteuern, dass wir sie kontrolliert beenden können?

Noch einmal der wichtigste Grundsatz für Goroutinen als Wiederholung von Kapitel 9.1: Beim Starten einer Goroutine muss auch immer deren Ende berücksichtigt werden. Bezüglich der Laufzeit gibt es zwei unterschiedliche Typen von Goroutinen:

- Goroutinen, die einen Loop beinhalten und auf ein oder mehrere unterschiedliche Ereignisse reagieren und somit *unendlich* laufen, und
- Goroutinen, die eine oder mehrere Anweisungen sequentiell abarbeiten und dann beendet werden.