

```

        // i erhöhen.

        i++
    }
}

```

### 3.11.3 Das böse goto

Der Dritte im Bunde ist das seit Langem verpönte **goto**, das immer gemeinsam mit einem Label zum Einsatz kommt. Dabei ist darauf zu achten, dass zwischen dem Sprung und dem Label keine neuen Variablen deklariert werden, weder implizit noch explizit. Eine Folge wäre sonst, dass ein späterer Zugriff nach dem Label auf eine nicht deklarierte Variable erfolgt. Und so führt das zwar nicht zu einem Übersetzungsfehler, dafür aber zu einem unerwarteten Ergebnis.

**Quelltext 3.92**  
*Unsichere Hüpfen*

```

a := 1

// Sprung über Zuweisung hinweg.

goto Target

b := 9

// Sprungziel.

Target:

b += a

fmt.Printf("A: %v / B: %v\n", a, b)

// A: 1 / B: 2

```

Und so gilt auch hier der Ratschlag aus anderen Sprachen: Auf den Einsatz von **goto** sollte verzichtet werden. Wenn er sich dennoch nicht vermeiden lässt, müssen Sie größte Vor- und Umsicht walten lassen.

## 3.12 Aufräumarbeiten

### 3.12.1 Verzögerte Ausführung mit defer

Eine besondere Funktionalität in Go dient der Verzögerung von Aktivitäten bis zum Verlassen einer Funktion. Dies klingt ungewöhnlich,

ist jedoch praktisch. Vielfach werden Funktionen so entwickelt, dass sie am Ende Ressourcen wieder freigeben oder Zustände sauber wiederherstellen müssen. Dies ist über unterschiedliche Ablaufpfade in der Funktion nur aufwendig zu erreichen, insbesondere zusammen mit dem zu jeder Zeit möglichen **return**.

Das hierfür vorgesehene Schlüsselwort ist **defer**, gefolgt von einem Funktions- oder Methodenaufruf. Es lässt sich mehrfach in einer Funktion anwenden. Hierbei werden jedes Mal die übergebenen Parameter ausgewertet, die Funktion hingegen nicht ausgeführt. Stattdessen werden die Funktionsaufrufe auf einem Stapel abgelegt und kommen dann am Ende der Funktion unabhängig von der Art der Beendigung zur Ausführung. Gibt die umgebende Funktion Werte zurück, werden diese zuerst ausgewertet, anschließend die verzögerten Funktionen. Danach erst erfolgt der Rücksprung. Auf diese Weise kann eine in der Funktion definierte anonyme Funktion für das **defer** innerhalb ihres Rumpfs benannte Rückgaben noch verändern.

```
// Rückgabe von MyType als String.
func (m *MyType) String() {
    // Sperre setzen und am Ende automatisch freigeben.

    m.mutex.Lock()
    defer m.mutex.Unlock()

    // String erzeugen.

    return fmt.Sprintf("{%v / %v / %d}",
        m.id, m.name, m.counter)
}
```

**Quelltext 3.93**  
defer für das  
Aufräumen

Dies ist schon so ein komfortables Hilfsmittel, insbesondere bei einer größeren Menge an Aktivitäten vor dem Verlassen der Funktion. Eine besondere Wichtigkeit erlangt der Mechanismus des **defer** jedoch im Zusammenhang mit *Laufzeitfehlern*. Auch hier kommt – ähnlich dem **finally** bei den Behandlung von *Exceptions* in Java – die mit **defer** verzögerte Funktion immer zum Einsatz und kann so Ressourcen wieder freigeben und Arbeiten ordnungsgemäß beenden.

### 3.12.2 Laufzeitfehler

Laufzeitfehler werden über die eingebaute Funktion **panic()** ausgelöst, sowohl durch das unterliegende Laufzeitsystem als auch in Bibliotheken. Der Funktion können beliebige Argumente als Details mitgeliefert werden. Diese stehen später für eine Analyse zur Verfügung. Für ih-

re Behandlung stehen jedoch keine umrahmenden *try* und *catch* zur Verfügung. Stattdessen kann innerhalb einer durch **defer** aufgerufenen Funktion geprüft werden, ob ein Laufzeitfehler aufgetreten ist. Die eingebaute Funktion hierfür ist die im folgenden Abschnitt vorgestellte Funktion **recover()**.

Wird nun ein Fehler mittels **panic()** ausgelöst, wird die aktuelle Programmausführung an Ort und Stelle unterbrochen. Diese Unterbrechung durchläuft den Stack bis zur obersten Funktion, wo das Programm mit einem Fehlerbericht abgebrochen wird. Einzig im Stack enthaltene und mit **defer** zurückgestellte Funktionen werden noch aufgerufen. Hier wird deutlich, warum die Funktion den Namen **panic()** trägt.

### 3.12.3 Rettungsarbeiten

Verfügen die unterbrochenen Funktionen über mittels **defer** zurückgestellte Funktionen, werden diese vor dem Abbruch ihrer umgehenden Funktionen wie oben beschrieben noch aufgerufen. In ihnen – jedoch nicht in hierin aufgerufenen Funktionen – kann nun mittels **recover()** geprüft werden, ob **panic()** aufgerufen wurde und welches Argument übergeben wurde. Gleichzeitig wird die rekursive Fehlerbehandlung an der Stelle des Aufrufs von **recover()** beendet und die reguläre Programmausführung im Gegensatz zum zuvor beschriebenen Standardverhalten fortgesetzt. Ein Aufruf von **recover()** außerhalb einer zurückgestellten Funktion ist zwar möglich, führt jedoch nur zu einer Rückgabe des Werts **nil**.

**Quelltext 3.94**  
*panic() und recover()*

```
// Funktion auf Daten aus MyType ausführen.
func (m *MyType) Do(f func([]byte)) (err os.Error) {
    // Zurückstellung der Fehlerbehandlung.

    defer func() {
        if e := recover(); e != nil {
            if oe, ok := e.(os.Error); ok {
                // Fehler ist ein os.Error.

                err = oe
            } else {
                // Fehler als String zurückgeben.

                err = os.NewError(fmt.Sprintf(e))
            }
        }
    }()
}
```

```
// Modifikation der Daten.  
  
f(m.data)  
  
return nil  
}
```

Das Beispiel zeigt, wie bei ordentlicher Ausführung `nil` als Fehler zurückgegeben wird. Im Fehlerfall wird hingegen geprüft, ob der Laufzeitfehler bereits vom Typ `os.Error` ist. In diesen Fall wird er direkt an den Aufrufer geliefert. Andernfalls wird er aus einer Darstellung des Fehlers als String erzeugt.

Ein transaktionales Verhalten wird auf diese Weise jedoch nicht sichergestellt. Vor einem Laufzeitfehler bereits modifizierte Daten werden nicht wieder zurückgesetzt. Wird dies benötigt, so muss auf einer Kopie gearbeitet werden, die erst nach erfolgreicher Verarbeitung die originalen Daten ersetzt.

## 3.13 Zusammenfassung

Googles Entwickler haben Go einfache und dennoch mächtige Strukturen mitgegeben. Der Wortschatz wurde gering gehalten, beispielsweise beim `for`, bleibt jedoch flexibel im Einsatz. Type Assertion und Type Switch machen den Einsatz der Reflection oftmals überflüssig, und `defer` erlaubt ein bequemes Aufräumen am Ende. Exceptions werden nicht vermisst, da `recover()` sowie die Rückgabe von Erfolgs- oder Fehlerwerten durch Funktionen eine gute Kontrolle erlauben. So haben Go-Programmierer stets eine gute und mächtige Kontrolle, ohne dass der Wortschatz überladen erscheint oder eine übertriebene Sprachvielfalt die Auswahl des richtigen Wegs erschwert.