Futures and Promises

In verschiedenen Programmiersprachen

Raphaele Licciardo (60559)
Prof. Dr. Sulzmann
14. Mai 2021

Futures and Promises

Bei der Programmierung stellt Future and Promise einen Platzhalter für ein unbekanntes Ergebnis dar, hauptsächlich weil die Berechnung noch nicht abgeschlossen ist. [Quelle 1]

Ein Future ist normalerweise das Ergebnis eines asynchronen Aufrufs einer Funktion oder Methode und kann verwendet werden, um sofort auf das Ergebnis zuzugreifen, sobald ein Ergebnis vorliegt. Diese Art der Programmierung ermöglicht eine weitgehend transparente Parallelisierung gleichzeitiger Prozesse. [Quelle 2]

Die zentrale Idee der Programmierung mit Futures ist, dass Futures als Parameter anderer Prozeduraufrufe übergeben werden können. Die Auswertung dieses Aufrufs kann dann gestartet werden, bevor zukünftige Ergebnisse verfügbar sind. Dies ermöglicht maximale Parallelität. Wenn der neue Anruf asynchron erfolgt, wird dies auch als Pipeline von Futures bezeichnet. Pipeline kann insbesondere in verteilten Anwendungen verwendet werden, um die Verzögerung der Kommunikation zwischen Prozessen zu minimieren. [Quelle 3]

Futures sind eine Konstruktion der asynchronen Kommunikation zwischen Prozessen. Konzeptionell bietet die Zukunft eine `Get`- oder `Join`-Funktion, die blockiert, bis das Ergebnis verfügbar ist und zurückkehrt. Je nach Implementierung kann die Wartezeit durch eine Zeitüberschreitung begrenzt werden, oder es können andere Funktionen verwendet werden, um den aktuellen Status abzufragen.

Wenn Futures direkt in eine Programmiersprache integriert ist, definieren diese normalerweise nur einen asynchronen Zuweisungsoperator, z. B. den Ausdruck „ `@= "`. Dies bedeutet: Es wird ein Prozess gestartet, um den Ausdruck auf der rechten Seite des Operators auszuwerten, und dieser der Variable `x` die Zukunft des Ergebnisses zuzuweisen. Wenn anschließend auf die

Variable x zugegriffen wird, wartet das System an dieser Stelle, bis das Ergebnis verfügbar ist. [Quelle 6]

Die Programmiersprache und Bibliothek, die Futures oder Versprechen unterstützt, ist CORBA (mit asynchronem Methodenaufruf (AMI)) und ab Version 5-Java (Concurrent Class Library) mit gleichzeitigen Dienstprogrammen. Seit ECMAScript 6 stellt JavaScript diese Konstrukte bereit (obwohl es immer noch in eingeschränkter Form vorliegt). Parallelität und Futures können auch in der Standardbibliothek von Standard C++ 11 verwendet werden. Andere Programmiersprachen, die Futures und Promises unterstützen, sind unter anderem Java, C++, .NET, Python, Swift, Dart, usw.. [Quelle 6]

In C# 5.0 und Visual Basic 2013 wird Future implizit über async verwendet und wartet. Die entsprechende zukünftige Klasse ist in der parallelen Erweiterung definiert, sodass sie auch in früheren Versionen und anderen Programmiersprachen verwendet werden kann. Sie können es jedoch bei Bedarf selbst implementieren. [Quelle 6]

Der folgende Pseudo Code zeigt eine beispielhafte Verwendung von Futures anhand dem genannten asynchronen Zuweisoperator @=

```
var x @= calc_x();
var y @= calc_y();
var z = calc_z();

var a := x + y + z;
```

Geht man nun davon aus, das die Berechnung von x und y noch andauert obwohl der Quellcode bereits an der Berechnung von a angekommen ist, würde es zu Fehler kommen. Anhand dem Future and Promise Prinzip wird auf die Berechnung von a gewartet. [Quelle 6]

Go

Für die Nebenläufige Programmiersprache Go gibt es keine offizielle Future und Promise Bibliothek wie es beispielsweise in C++ der Fall ist. Schließlich braucht man dies auch nicht, dank der Channels können Future und Promise Objekte einfach nachgebildet werden. [Quelle 5]

Durch die Go internen Kanäle ist dies aber kein großer Aufwand. So kann Beispielsweise ein Future wie folgt aufgebaut werden.

```
func future(function func() (int, bool)) Future {
    channel := make(chan futureType)
    go func() {
        value, status := function()
        fType := futureType{value, status}
        for {
            channel <- fType
        }
    }()
    return channel
}
```

[Quelle 5]

Das hier in Go implementierte Future führt eine Asynchrone Berechnung aus, um einen Wert zu erzeugen. Sobald diese Berechnung abgeschlossen ist, wird dieser dank des Kanals an den Future gebunden. Hier wird ebenso ein boolescher Rückgabeparameter angelegt, um darzustellen, ob die Berechnung erfolgreich war oder fehlgeschlagen ist. Ein Fehlschlag kann ein erkannter Rechenregel-Bruch sein oder ein einfacher Timeout einer HTTP Anfrage.

```
func (future Future) get() (int, bool) {
    fType := <-future
    return fType.value, fType.status
}
```

[Quelle 5]

Die Get Methode fragt den an den Future beziehungsweise Kanal gebunden Wert ab. Falls dieser noch nicht vorhanden ist, wird entsprechend blockiert.

```
func (future Future) onSuccess(callback func(result int)) {
    go func() {
        value, status := future.get()
        if status {
            callback(value)
        }
    }()
}
```

[Quelle 5]

Im Gegensatz zu der `get()` Methode ist der `onSuccess()` Aufruf nicht blockierend. Als Argument nimmt diese Funktion eine Callback Funktion entgegen. Hierbei wird der an den Future gebundene Wert verarbeitet. Natürlich sobald dieser verfügbar ist. Diese Methode trifft nur ein, falls die Berechnung auch erfolgreich war.

```
func (future Future) onFailure(callback func()) {
    go func() {
        _, status := future.get()
        if !status {
            callback()
        }
    }()
}
```

[Quelle 5]

Auch der `onFailure()` Aufruf ist nicht blockierend. Dieser nimmt ebenso eine Callback Funktion entgegen. Analog zu oben, trifft diese nur ein, wenn die Berechnung des Ergebnisses fehlgeschlagen ist.

C++

C++ ist eine Programmiersprache von Bjarne Stroustrup aus 1979 als eine Erweiterung für die Programmiersprache C. C++ ermöglicht sowohl die effiziente und maschinennahe Programmierung als auch eine Programmierung auf hohem Abstraktionsniveau. Der Standard definiert auch eine Standardbibliothek, zu der verschiedene Implementierungen existieren. Eine davon ist die `<future>` Bibliothek, welche einen Mechanismus bietet, um auf das Ergebnis von asynchronen Operationen zuzugreifen. [Quelle 7]

Im Folgenden werden verschiedenste Berechnungen durchgeführt. Solche Berechnungen können durchaus viel Rechenleistung und somit Zeit in Anspruch nehmen. Ebenso werden meistens solche Berechnungen für weitere Rechnungen benötigt. Auf den ersten Blick ist Nebenläufigkeit das Tool der Wahl. Dauert die Rechnung jedoch länger wie erwartet kann es zu Problemen kommen. In dieser C++ Implementierung wird dieses Problem umgangen.

Zunächst muss man anhand folgender Zeile ein Versprechen initialisieren. Mit diesem Versprechen kann ein Wert trotz zeitlicher Verzögerungen gesetzt werden.

```
std::promise<int> promise;
```

[Quelle 7]

Aus diesem Promise wird nun ein Future erstellt. Dieses Future kann mehrere Aufgaben haben, unter anderem kann daraus das Promise eingelöst und der Wert eingeholt werden. Ebenso wartet das Future auf eine Art Benachrichtigung des Promise ob die Berechnung fertig ist.

```
std::future<int> result = promise.get_future();
```

[Quelle 7]

Der Thread benötigt als erstes Argument ein sogenanntes `Callable`. Ein solches kann Beispielsweise eine Methode sein, welche eine Berechnung beinhaltet. Dies kann wie folgt umgesetzt werden.

```
void add(std::promise<int>&& promise, int a, int b) {
    promise.set_value(a+b);
}
```

Es wird jeweils ein Promise und die zu verrechnenden Zahlen benötigt. Das Promise wird hierbei durch das `std::move` der `<Future>` Bibliothek bereitgestellt. Unabhängig von der Dauer der Berechnung kann nun die Berechnung gestartet werden.

```
std::thread thread(calc, std::move(promise), a, b);
```

C++ stellt dank der `<Future>` Bibliothek nicht nur Future und Promises zur Verfügung, sondern auch sogenannte `std::shared_future`. Diese Erlaubt folgende Funktionalitäten bzw. Eigenschaften:

- Promises unabhängig der zugehörigen Futures abfragen
- Gleiche Schnittstelle wie ein `std::future` Objekt.
- Kann aus einer bestehenden Zukunft erzeugt werden.
  - Durch `future.share()`
- Wird aus einem beliebigen Promise erzeugt: [Quelle 7]

```
std::shared_future<int> result = promise.get_future()
```

Jedoch ist deren Verwaltung etwas komplizierter. Im Folgenden wird darauf näher eingegangen. Zunächst ist der Anfang recht identisch. Man muss ein Promise initialisieren

```
std::promise<int> promise;
```

Die Initialisierung der Shared Future ist wie bereits angeschnitten minimal verschieden. Aus dem Promise wird ähnlich zu oben ein Future generiert, jedoch wird es hier in einer speziellen Shared Future Variable gespeichert.

```
std::shared_future<int> result = promise.get_future();
```

Auch die Erstellung des Threads ist identisch:

```
std::thread thread(add, std::move(promise), a, b);
```

Besonders zu beachten sind nun die Shared Futures Threads:

```
std::thread shared_thread_1(req, result);
std::thread shared_thread_2(req, result);
// [...]
std::thread shared_thread_n(req, result);

shared_thread_1.join();
shared_thread_2.join();
// [...]
shared_thread_n.join();
```

Ähnlich wie eine herkömmliche Future arbeitet auch das Shared Future Objekt, mit dem Unterschied, dass mehrere Threads auf denselben gemeinsamen Zustand warten dürfen. Im Gegensatz zur Future, das nur verschiebbar ist (sodass nur eine Instanz auf ein bestimmtes asynchrones Ergebnis verweisen kann), ist die Shared Future kopierbar und mehrere Shared Future Objekte können auf denselben gemeinsamen Zustand verweisen. Der Zugriff auf denselben gemeinsamen Zustand von mehreren Threads aus ist sicher, wenn jeder Thread dies über seine eigene Kopie eines Shared Future Objekte erledigt. [Quelle 7]

Jeder Eltern Thread muss sich gewissermaßen auch um seine Kinder Threads kümmern. Beispiele kann der Eltern Thread warten bis der Kinder Thread fertig ist oder dieser sich abgekoppelt hat. Diese Eigenschaft ist recht bekannt, gilt jedoch nicht für `std::async`. Hierbei muss ein Eltern Thread sich nicht um die Kinder Threads kümmern. Diese Eigenschaft wird ebenfalls in einer Variante des Future und Promise eingebunden. [Quelle 7]

Die sogenannte Fire und Forget Variante erzeugt mit der genannten Asynchronität ein spezielles Promise. Diese warten in dem Destruktor auf das Einlösen des Versprechens. Dadurch muss sich der Eltern Thread auch nicht um die Kinder Threads kümmern. [Quelle 7]

Somit kann man `std::future` Objekt als Fire und Forget Job ausgeführt werden. Denn das `std::async std::future` Objekt ist an keine Variable gebunden wie es bei der herkömmlichen Variante der Fall ist. Somit kann mit dem gleichen Future Objekt sowohl gewartet als auch ein Ergebnis geholt werden. In der folgenden Variante ist eine beispielhafte Implementierung um ein solches Prinzip zu erreichen:

```
void blocking() {
    std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "Blocking> Thread 1" << std::endl;
    });
    std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "Blocking> Thread 2" << std::endl;
    });
    std::cout << "Blocking> Main Thread" << std::endl;
}
```

Beide Threads warten in dessen Destruktor auf die Beendigung der Arbeit. Ein anderes Wort für ewiges warten ist blockieren. Und genau dies passiert hier. Es werden zwei Versprechen in den eigenen Threads gestartet, die resultierenden Fire und Forget Futures. Diese blockieren den Destruktor so lange bis das entsprechende Versprechen eingelöst ist. Das Ergebnis ist hierbei der Verlust des Future und Promise Prinzip, die Ausgabe des Programmes ist wie folgt:

```
Blocking> Thread 1
Blocking> Thread 2
Blocking> Main Thread
```

Obwohl im Main Thread zwei neue Promise Threads erzeugt wurden, die jeweils in eigenen Threads ausgeführt werden, wird dennoch jeder Thread nacheinander ausgeführt. Demnach ist der erste Thread mit der längsten Arbeitszeit auch als erstes fertig.

Im folgenden Code Beispiel wird der Future an eine zugehörige Variable gebunden. Das Blockieren in dessen Destruktor findet erst statt, wenn die Variable ihre Gültigkeit verliert.

```
void not_blocking() {
    auto first = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "Not Blocking> Thread 1" << std::endl;
    });
    auto second = std::async(std::launch::async, [] {
    std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "Not Blocking> Thread 2" << std::endl;
    });
    std::cout << "Not Blocking> Thread Main" << std::endl;
}
```

Dadurch zeigt die Ausgabe des Programmes aus den gewünschten Output mit einer kleinen aber wichtigen Veränderung:

```
Not Blocking> Thread Main
Not Blocking> Thread 2
Not Blocking> Thread 1
```

Die Ausgabe entspricht dem intuitiven Gedanken bei einer Nebenläufigen Ausführung von Threads. Schließlich besitzen die Kinder Threads (Future 1 und 2) jeweils deren Gültigkeit bis zum Ende des Eltern Threads (Main Thread). Somit ist der Thread mit der kleineren Arbeitszeit schneller fertig.

In diesem Beispiel werden Futures recht speziell eingesetzt. Zum einen werden diese an keine explizite Variable gebunden und zum anderen werden diese nicht dazu verwendet das Ergebnis mit einem blockierenden get() oder `wait()` anzufordern. Jedoch wird nur unter dieser speziellen Rahmenbedingung erst das besagte Phänomen sichtbar. Ein Blockierender Destruktor bei einem Asynchron erstellten Fire and Forget Future der an keine Variable gebunden ist.

Quellenangaben

1. Promise. MDN. Mozilla Foundation
2. Asynchronous programming. MSDN. Microsoft
3. Futures. MSDN, Parallel Programming with Microsoft .NET. Microsoft
4. Dan Vanderboom: Concurrency & Coordination With Futures in C#. In: Critical Development
5. Autonome Systeme Vorlesung, Prof. Dr. Sulzmann
6. Wikipedia, Futures and Promises
7. cplusplus, Futures and Promises
8. cplusplus, Shared Future