# Guice による依存性注入

## ボイラープレート・コードを減らし、テストしやすいコードを作成する

Nicholas Lesiecki
Software engineer
Google

2008年 12月 09日

Guice は Google によるオープンソースの Java™ 開発用依存性注入フレームワークです。Guice を利用すると独自のファクトリーを作成する苦労をしなくてすむため、テストが容易になり、またプログラムのモジュール性を高めることができます。この記事では Nicholas Lesiecki が、皆さんがアプリケーションを作成する中で Guice を使いこなせるように、Guice の最も重要な概念について説明します。

Guice は DI (Dependency Injection: 依存性注入) のためのフレームワークです。私は長年、DI を使用するように開発者達に勧めてきましたが、それは DI を使用することによって保守性、テスト性、そして柔軟性を高めることができるためです。私は Guice に対するエンジニア達の反応を見つめる中で、新しい技術の採用をプログラマーに納得させるための最良の方法は、その技術の導入を極めて容易なものにすることであるということを学んできました。Guice によって DI は非常に容易になり、その結果 Google では DI が普通に使われるようになりました。この記事では、それと同じことが皆さんにも起こるように、Guice についてとてもわかりやすく説明します。

### この記事は Guice についてのツアーであり、議論の場ではありません

Guice はいくつもの画期的な DI フレームワークに続いて登場しました。(そうしたフレームワークの 1 つである PicoContainer のページには DI フレームワークの歴史と相関関係が説明されています。「参考文献」を参照してください)。Guice は後になって登場したため、どのフレームワークが優れているのか、さらに別の DI フレームワークが必要なのかという議論に火がつきました。どの技術を選択する場合も同じですが、それぞれのライブラリーには長所と短所があります。私は Guice によって新たなものが議論の場に持ち込まれたと思いますが、この記事ではそうした議論に加わることよりも Guice の特徴を説明することに焦点を絞ります。(Web で「guice vs spring」と検索すると、いくつかの活発な議論を見るけることができます。)

## Guice 2.0 ベータ版

この記事の執筆時点で Guice チームは Guice 2.0 のための作業を懸命に続けており、2008年の終わりまでに Guice 2.0 がリリースされることが期待されます。Google Code のダウンロード・サイトには初期のベータ版が置いてあります (「参考文献」を参照)。これは良い知らせです。なぜな

ら、Guice チームは Guice コードを使いやすく、そして理解しやすくする機能を追加したからです。このベータ版には一部の機能が欠けているため最終バージョンではありませんが、安定しており、質の高いバージョンです。実際、Google はこのベータ版を製品ソフトウェアに使用しています。私は皆さんにも、同じようにこのベータ・バージョンを本番用ソフトウェアに使用してみるようにお勧めします。この記事は Guice 2.0 に焦点を絞って書いており、Guice の新機能をいくつか説明しますが、1.0 で非推奨となった機能についてはほとんど触れません。ここで説明する機能に関しては、現在のベータ版と最終リリースとの間で変わることはないと Guice チームが確約してくれています。

皆さんが既に DI を理解しており、なぜ DI を支援するフレームワークが必要であるかを知っている場合には、次のセクションをとばして「Guice による基本的な注入」に進んでください。それ以外の方は DI によるメリットを説明する次のセクションを読んでください。

## DI によるメリット

まず、例を説明することから始めましょう。例えば私がスーパーヒーローのアプリケーションを作成しており、Frog Man (カエル男) という名前のヒーローを実装しているとします。リスト 1 には、そのコードと最初のテストが含まれています。(ユニット・テストを作成する価値は説明するまでもないと思います。)

### リスト 1. 基本となるヒーローと、そのテスト

```
public class FrogMan {
  private FrogMobile vehicle = new FrogMobile();
  public FrogMan() {}
  // crime fighting logic goes here...
}

public class FrogManTest extends TestCase {
 public void testFrogManFightsCrime() {
    FrogMan hero = new FrogMan();
    hero.fightCrime();
    //make some assertions...
  }
}
```

すべては順調に見えますが、テストを実行しようとすると、リスト 2 の例外がスローされてしまいます。

### リスト 2. 依存関係は問題を起こしがちです

```
java.lang.RuntimeException: Refinery startup failure.
  at HeavyWaterRefinery.<init>(HeavyWaterRefinery.java:6)
  at FrogMobile.<init>(FrogMobile.java:5)
  at FrogMan.<init>(FrogMan.java:8)
  at FrogManTest.testFrogManFightsCrime(FrogManTest.java:10)
```

どうやら `FrogMobile` (カエル車) は `HeavyWaterRefinery` (重水精製所) を構築するようです。ここで仮に、テストの中で `HeavyWaterRefinery` を構築する方法はない、としましょう。本番環境では `HeavyWaterRefinery` を構築することができても、単にテストのためだけに第 2 の `HeavyWaterRefinery` を構築する許可を得ることはできません。現実の世界では、重水を精製することはないかもしれませんが、リモート・サーバーや大規模なデータベースに依存する可能性は

あり得ます。このいずれの場合にも同じ原則が当てはまるのです。つまり依存関係があるものを立ち上げるのは難しく、相互作用にも時間がかかり、テストでも予想以上に失敗する原因となります。

## DI の登場

この問題を回避するための方法として、インターフェース (例えば `Vehicle` (乗り物) など) を作成し、`FrogMan` クラスがコンストラクターの引数として `Vehicle` を受け付けるようにする方法があります (リスト 3)。

## リスト 3. インターフェースに依存してインターフェースに依存性を注入させる

```
public class FrogMan {
  private Vehicle vehicle;

  public FrogMan(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
  // crime fighting logic goes here...
}
```

このイディオムが DI の本質を示しています。つまり依存関係を作成する (あるいは静的参照を使う) 代わりに、クラスがインターフェースを参照することで依存関係を受け付けるようにするのです。リスト 4 は DI によってテストがどれほど容易になるかを示しています。

## リスト 4. 問題を起こしがちな依存関係を使わず、モックを使ってテストする

```
static class MockVehicle implements Vehicle {
  boolean didZoom;

  public String zoom() {
    this.didZoom = true;
    return "Mock Vehicle Zoomed.";
  }
}

public void testFrogManFightsCrime() {
  MockVehicle mockVehicle = new MockVehicle();

  FrogMan hero = new FrogMan(mockVehicle);
  hero.fightCrime();

  assertTrue(mockVehicle.didZoom);
  // other assertions
}
```

このテストでは手動で作成したモック・オブジェクトを `FrogMobile` の置き換えとして使っています。DI を使うと、テスト用に `HeavyWaterRefinery` を立ち上げるという面倒な作業が必要なくなり、`FrogMobile` について知る必要もなくなります。テストに必要なものは `Vehicle` インターフェースのみです。DI によってテストが容易になるだけではなく、コード全体のモジュール性と保守性が向上します。また、`FrogMobile` を `FrogBarge` (カエル船) に切り換えたい場合にも `FrogMan` を変更する必要がありません。`FrogMan` はインターフェースのみに依存するのです。

ただし、落とし穴があります。皆さんが私と同じなら、DI についての説明を初めて聞くと、「それならば、`FrogMan` を呼び出すものはすべて、`FrogMobile` (そして `HeavyWaterRefinery` や

`HeavyWaterRefinery` の依存関係等々) について知らなければならない」と思うかもしれません。もしそうだったとしたら、DI は定着しなかったでしょう。呼び出し側に負担を強制するのではなく、ファクトリーを作成して、オブジェクトの作成やオブジェクトの依存関係の作成をファクトリーに管理させることができるのです。

ファクトリーを使うとなると、フレームワークが登場します。ファクトリーには面倒で繰り返しの多いコードが大量に必要です。最良の場合でもファクトリーはプログラム作成者 (そしてプログラムを読む人) を悩ませ、最悪の場合にはあまりにも面倒なため作成されることすらありません。Guice や他の DI フレームワークはオブジェクトの作成作業を柔軟に構成できる「スーパー・ファクトリー」の役割を果たします。フレームワークを構成する作業は独自のファクトリーを作成するよりも遥かに容易です。その結果、プログラマーは DI のスタイルでコードを作成するようになります。これにより、より多くのテストが行われ、より良いコードが作成され、そしてプログラマーは幸せになります。

## Guice による基本的な注入

上の説明から、DI によって設計の価値が高まること、またフレームワークを使うことで設計がさらに楽になることを理解できたと思います。では Guice について詳しく調べてみましょう。まず `@Inject` アノテーションとモジュールから始めることにします。

### `@Inject` を使ってクラスに注入するように Guice に指示する

`FrogMan` と、Guice での `FrogMan` の唯一の違いは `@Inject` です。リスト 5 はアノテーションを持つ `FrogMan` のコンストラクターです。

### リスト 5. `@Inject` を使って `FrogMan` に注入する

```
@Inject
public FrogMan(Vehicle vehicle) {
  this.vehicle = vehicle;
}
```

一部のエンジニアはクラスに `@Inject` を追加するという考え方を嫌います。クラスは DI フレームワークについてまったく知らない方が好ましいと彼らは考えるのです。それは妥当なことなのですが、私は同意はしません。依存関係に関するアノテーションはそれほど大げさなものではありません。`@Inject` タグが意味を持つのは、クラスを作成するように Guice に指示する場合のみです。`FrogMan` を作成するように Guice に指示する場合以外、このアノテーションがコードの動作に影響することはありません。アノテーションという便利な手掛かりによって、Guice はクラスの作成に参加することができます。しかしアノテーションを使うためにはソース・レベルでのアクセスが必要になります。アノテーションを使いたくない場合、あるいは自分ではソースを管理しないオブジェクトを作成するために Guice を使っている場合、Guice にはアノテーションの代わりとなるメカニズムが用意されます (この記事の後の方にある囲み記事、「プロバイダー・メソッドの、その他の使い方」を見てください)。

### どの依存関係が必要なのかを Guice に指示する

Guice は、ヒーローが `Vehicle` を必要とすることを理解できたので、今度はどの `Vehicle` を提供するのかを知る必要があります。リスト 6 には `Module` が含まれています。`Module` は、どの実装にはどのインターフェースを使うのかを Guice に指示するために使う特別なクラスです。

## リスト 6. `HeroModule` によって `Vehicle` を `FrogMobile` にバインドする

```
public class HeroModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Vehicle.class).to(FrogMobile.class);
  }
}
```

モジュールは 1 つのメソッドを持つインターフェースです。Guice がモジュールに渡す `Binder` によって、オブジェクトの作成方法を Guice に指示することができます。バインダーの API はドメイン特化言語を形成します (「参考文献」を参照)。このミニ言語によって、例えば `bind(X).to(Y).in(Z)` のように表現力に富んだコードを作成することができます。バインダーによって実現できることについてのさらなる例は、この先を読み進めていくなかで紹介しています。`bind` を呼び出すたびにバインディングが作成され、Guice はそうした一連のバインディングを使って注入に関する要求を解決するのです。

## `Injector` によるブートストラップ

次に、`Injector` クラスを使って Guice を起動します。通常はプログラムの非常に初期の段階でインジェクターを作成する必要があります。そうすれば Guice が大部分のオブジェクトを作成してくれます。リスト 7 には `Injector` を使ってヒーローの冒険を開始する main プログラムの例をあげています。

## リスト 7. `Injector` を使ってアプリケーションを起動する

```
public class Adventure {
  public static void main(String[] args){
    Injector injector = Guice.createInjector(new HeroModule());
    FrogMan hero = injector.getInstance(FrogMan.class);
    hero.fightCrime();
  }
}
```

インジェクターを作成するためには Guice クラスの `createInjector` を呼び出します。このとき `createInjector` にモジュールのリストを渡すと、`createInjector` はそれらのモジュールを使って`createInjector` 自身を構成します。(この例には 1 つのモジュールしかありませんが、悪者を構成する `VillainModule` を追加することもできます。) インジェクターを作成できたら、`getInstance` を使ってオブジェクトを取得するようにインジェクターに要求し、返して欲しい `.class` を渡します。(洞察力の鋭い読者は Guice に対して `FrogMan` に関する指示をする必要がないことに気付くでしょう。具象クラスを要求する場合でそのクラスが `@Inject` コンストラクターを持つ場合、または引数のない public コンストラクターを持つ場合には、`bind` を呼び出さなくても Guice がそのクラスを作成してくれることがわかります。)

これは Guice を使ってオブジェクトを作成するための第 1 の方法、つまり明示的に要求する方法です。しかしこれをブートストラップ・ルーチン以外の場所で行ってはなりません。もっと良い、そしてもっと容易な方法は、依存性の注入、依存性の依存性の注入、等々を Guice に行わせる方法です。(「it's turtles all the way down (どこまでも亀が続く)」(「参考文献」を参照) という言い習わしにあるとおりです。) これは最初、不安に思えるかもしれませんが、しばらくすると慣れるものです。リスト 8 は `FrogMobile` に `FuelSource` (燃料源) を注入する例を示しています。

### リスト 8. `FrogMobile` が `FuelSource` を受け付ける

```
@Inject
public FrogMobile(FuelSource fuelSource){
  this.fuelSource = fuelSource;
}
```

これはつまり、アプリケーションが `FrogMan` を取得しようとすると、そのアプリケーションとインジェクターとのやり取りは一度しか行われていないにもかかわらず、Guice は `FuelSource` と `FrogMobile`、そして最後に `FrogMan` まで作成する、ということです。

もちろん、アプリケーションの `main` ルーチンを常に制御できるとは限りません。例えば多くの Web フレームワークでは、「アクション」や「テンプレート」、その他の出発点となるものが魔法のように自動的に作成されてしまいます。しかし通常はフレームワーク用のプラグインや手動で作成した独自のコードを使うことで、Guice を挿入するための場所を見つけることができるはずです。(例えば Guice プロジェクトは Struts のアクションを Guice を使って構成できる Struts 2 用のプラグインをリリースしています。(「参考文献」を参照))

## 他の形式による注入

ここまではコンストラクターに `@Inject` を適用した例を説明しました。Guice はこのアノテーションを見つけると、そのコンストラクターの引数を調べ、それぞれの引数に対して構成済みのバインディングを見つけようとします。これはコンストラクター注入として知られています。Guice のベスト・プラクティス・ガイドによると、依存関係を要求する方法としてコンストラクター注入が推奨されています。しかしコンストラクター注入が唯一の方法というわけではありません。リスト 9 は `FrogMan` クラスを構成する別の方法を示しています。

### リスト 9. メソッド注入

```
public class FrogMan{
  private Vehicle vehicle;

  @Inject
  public void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
//etc. ...
```

ここでは注入されたコンストラクターが取り除かれており、代わりに `@Inject` でタグ付けされたメソッドがあることに注目してください。Guice は私のヒーローを作成した直後にこのメソッドを呼び出します。これは Spring フレームワークのファンにとっては「セッター注入」に見えるかもしれません。しかし、Guice が注目するのは `@Inject` のみであり、メソッドには任意の名前をつけることができ、またメソッドは複数のパラメーターを取ることができます。メソッドをパッケージで保護することもできれば private にすることもできます。

private メソッドにアクセスできるという Guice の決定に問題があると思う人は、リスト 10 を見てください。ここでは `FrogMan` がフィールド注入を使っています。

### リスト 10. フィールド注入

```
public class FrogMan {
  @Inject private Vehicle vehicle;
  public FrogMan(){}
//etc. ...
```

この場合も、Guice が注目するのは `@Inject` アノテーションのみです。Guice はアノテーションが付いたすべてのフィールドを見つけ、そこに適切な依存性を注入しようとします。

### どの方法が最適か

3 つのバージョンの `FrogMan` は同じ動作をします。つまり `FrogMan` が作成されると、Guice は適切な `Vehicle` を注入します。しかし私は Guice の作成者達と同じように、コンストラクター注入を好みます。紹介した 3 つのスタイルを簡単に分析すると次のようになります。

- **コンストラクター注入**は単純です。Java 技術ではコンストラクターの呼び出しが保証されているため、(Guice がオブジェクトを作成するか否かによらず) 初期化されていない状態のオブジェクトを受け取ることを心配する必要がありません。またフィールドには `final` を指定することもできます。
- **フィールド注入**は、特にフィールドを `private` と指定した場合、テストがしにくくなります。これは DI の主要な目的の 1 つに反することになります。非常に限定された状況以外、フィールド注入を使用すべきではありません。
- **メソッド注入**は、クラスのインスタンス化を制御しない場合に便利です。また何らかの依存関係を必要とするスーパークラスがある場合にも便利です。(これはコンストラクター注入では困難です。)

## 実装を選択する

例えば、アプリケーションの中に複数の `Vehicle` があるとしましょう。Frog Man と同様のヒーロー (ヒロイン) である Weasel Girl (イタチ娘) は `FrogMobile` を運転することができません。また、`WeaselCopter` (イタチ・ヘリコプター) に依存関係をハードコーディングすることは避けたいものです。この場合、Guice では依存関係にアノテーションを付けることができるため、この問題を解決することができます。リスト 11 では Weasel Girl がもっと高速な移動手段を要求しています。

### リスト 11. アノテーションを使って特定の実装を要求する

```
@Inject
public WeaselGirl(@Fast Vehicle vehicle) {
  this.vehicle = vehicle;
}
```

リスト 12 では、`HeroModule` (ヒーロー・モジュール) はバインダーを使うことによって `WeaselCopter` が「fast (高速)」であることを Guice に伝えています。

### リスト 12. `Module` の中でアノテーションについて Guice に指示する

```
public class HeroModule implements Module {
 public void configure(Binder binder) {
    binder.bind(Vehicle.class).to(FrogMobile.class);
    binder.bind(Vehicle.class).annotatedWith(Fast.class).to(WeaselCopter.class);
  }
}
```

ここで、どんな種類の vehicle を必要としているのかを記述するアノテーションとして、あまりにも実装に直結しすぎる言葉 (`@WeaselCopter`) ではなく抽象的な言葉 (`@Fast`) を選択していることに注目してください。意図する実装がアノテーションによって明確になりすぎると、それを読む人の頭の中で暗黙的な依存関係が作られてしまいます。もし、`@WeaselCopter` を使用している場合に Weasel Girl が Wombat Rocket (ウォンバット・ロケット) を借りたとすると、そのコードを読むプログラマーやデバッグするプログラマーを混乱させる可能性があります。

`@Fast` アノテーションを作成するためには、リスト 13 に示すボイラープレート・コードをコピーする必要があります。

### リスト 13. このコードをコピー・アンド・ペーストしてバインディング・アノテーションを作成する

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@BindingAnnotation
public @interface Fast {}
```

`BindingAnnotations` を大量に作成すると、このようなちょっとしたファイルを大量に作成することになりますが、各ファイルの違いはアノテーションの名前の違いのみです。これが非常に気になる場合、あるいは何らかの簡単なプロトタイピングを行いたい場合には、Guice に組み込みの、ストリング属性を受け付ける `@Named` アノテーションを使うことを検討することができます。リスト 14 はこの方法を示しています。

### リスト 14. カスタム・アノテーションの代わりに `@Named` を使う

```
// in WeaselGirl
@Inject
public WeaselGirl(@Named("Fast") Vehicle vehicle) {
  //...
}

// in HeroModule
binder.bind(Vehicle.class)
  .annotatedWith(Names.named("Fast")).to(WeaselCopter.class);
```

この方法は有効ですが、名前がストリングの中にあるため、コンパイル時のチェックとオートコンプリートをあきらめる必要があります。全体として考えると、私はむしろ独自のアノテーションを作成したいと思います。

アノテーションをまったく使いたくない場合にはどうすればよいのでしょう。クラスに `@Fast` または `@Named("Fast")` を追加するだけでも、そのクラスはクラスの一部を自ら構成することになります。それも気になるのであれば、この先を読んでください。

## プロバイダー・メソッド

皆さんはすべての冒険に Frog Man を送り出すことに嫌気がさしています。新しい冒険ごとにヒーローをランダムに登場させたいものです。Guice のデフォルトのバインダー API では、「呼び出しごとに別の実装に `Hero` クラスをバインドする」というような呼び出しはできませんが、新しい `Hero` をすべて特別なメソッドを使って作成するように Guice に指示することはできます。リス

ト 15 は `HeroModule` に追加された新しいメソッドであり、このメソッドは特別な `@Provides` アノテーションが使われています。

### リスト 15. プロバイダーを使ってカスタムの作成ロジックを作成する

```
@Provides
private Hero provideHero(FrogMan frogMan, WeaselGirl weaselGirl) {
  if (Math.random() > .5) {
    return frogMan;
  }
  return weaselGirl;
}
```

Guice は `Module` の中にある `@Provides` アノテーションが付いたすべてのメソッドを自動的に発見します。Guice は `Hero` の戻り型を基に、ヒーローが要求された場合にはこのメソッドを呼び出してヒーローを提供するのだということを判断します。プロバイダー・メソッドにロジックを追加することで、オブジェクトを作成することもでき、オブジェクトをランダムに選択することもでき、キャッシュからオブジェクトを見つけることもでき、あるいは他の方法でオブジェクトを取得することもできます。プロバイダー・メソッドは他のライブラリーを Guice モジュールに統合する素晴らしい方法です。またプロバイダー・メソッドは Guice 2.0 の新機能でもあります。(Guice 1.0 の方法ではカスタムのプロバイダー・クラスを作成しましたが、この方法は洗練されておらず、冗長でした。Guice 1.0 を使うことを決めた場合には、ユーザーズ・ガイドに古い方法の説明があります。またこの記事に付属しているサンプル・コードにはカスタムのプロバイダーがあるので、それを調べることもできます。)

Guice はリスト 15 のプロバイダー・メソッドに適切な引数を注入します。これはつまり、Guice がバインディングのリストから `WeaselGirl` と `FrogMan` を見つけるため、プロバイダー・メソッドの中で手動によって `WeaselGirl` と `FrogMan` を作成する必要がないということです。これは先ほど触れた「it's turtles all the way down」の原則を示しています。Guice モジュールそのものを構成する際にも、Guice が依存関係を提供してくれるのです。

#### プロバイダー・メソッドの、その他の使い方

Guice に付属しているアノテーションを使いたくない場合、あるいはそうしたアノテーションを使うことができない場合 (例えばサードパーティーのクラスを作成している場合など) には、プロバイダー・メソッドを使うことでスマートに問題を解決することができます。プロバイダー・メソッドはモジュールの中にあるため、アノテーションを付けてもソースの他の部分からアノテーションが見えてしまう心配はありません。例えばリスト 16 には (サードパーティーのヒーローである) `BadgerBoy` (アナグマ少年) のためのプロバイダー・メソッドが含まれています。

### リスト 16. プロバイダー・メソッドを使うとソースにアノテーションを付けずにすむ

```
@Provides @Inject
private BadgerBoy provideBadgerBoy(WeaselCopter copter) {
  return new BadgerBoy(copter);
}
```

このプロバイダー・メソッドを使うと、Guice を使って `BadgerBoy` を構成することができ、さらにはどの `Vehicle` が必要かを選択することもでき、そのために `BadgerBoy` を変更する必要はまっ

たくありません。これはつまり、使いたくなければ `@Inject` を使う必要はなく、あるいは `@Fast` のようなバインディング・アノテーションを使う必要もないということです。つまりプロバイダー・メソッドを使って依存関係を選択することができるのです。

どの手法を選択するかは、個人的な好みや、サードパーティーのクラスを使うかどうか、クラスとその依存関係とをどれくらい緊密に結びつけるかに依存します。プロバイダー・メソッドはまったく手のかからない方法です。依存関係にアノテーションを付けると、構成情報とクラスが少し近くなるためソースが理解しやすくなる一方、柔軟性とテストのしやすさは保持されます。おそらく皆さんのアプリケーションでは、状況に応じて両方の方法で依存関係を指定することになるでしょう。

## 依存関係の代わりに `Provider` を要求する

例えば `Saga` (冒険物語) という 1 つのストーリーの中に複数のヒーローが欲しいとしましょう。`Hero` を注入するように Guice に要求しても 1 人の Hero しか得られません。しかし「provider of heroes (ヒーローのプロバイダー)」を要求すると、好きなだけの数のヒーローを作成することができます (リスト 17)。

### リスト 17. プロバイダーを注入することでインスタンス化を制御する

```
public class Saga {
  private final Provider<Hero> heroProvider;

  @Inject
  public Saga(Provider<Hero> heroProvider) {
    this.heroProvider = heroProvider;
  }

  public void start() throws IOException {
    for (int i = 0; i < 3; i++) {
      Hero hero = heroProvider.get();
      hero.fightCrime();
    }
  }
}
```

またプロバイダーを使用すると、実際に Saga が起動するまでヒーローの取得を遅らせることができます。これは時刻やコンテキストによって大きく変化するデータに依存するヒーローの場合に便利です。

`Provider` インターフェースには `get<T>` という 1 つのメソッドがあります。提供されたオブジェクトにアクセスするためには、単純にこのメソッドを呼び出します。毎回新しいオブジェクトを取得するのかどうか、またそのオブジェクトをどのように構成するかは、Guice がどのように構成されたのかに依存します。(「スコープ」に関する次のセクションでは、シングルトンやその他の存続期間の長いオブジェクトの詳細を説明しています。) ここでは Guice が `@Provides` メソッドを使っていますが、これは新しい `Hero` を作成するための方法として `@Provides` メソッドが登録されているためです。これはつまり、Saga には 3 人のランダムなヒーローが混在する必要があるということです。

Provider をプロバイダー・メソッドと混同してはいけません。(Guice 1.0 では両者を区別することは非常に困難でした)。`Saga` はカスタムの `@Provides` メソッドからヒーローを取得します

が、Guice がインスタンス化したすべての依存関係の `Provider` を要求することもできます。必要な場合には、`FrogMan` のコンストラクターをリスト 18 のように書き直すことができます。

### リスト 18. 依存関係の代わりに `Provider` を要求する

```
@Inject
public FrogMan(Provider<Vehicle> vehicleProvider) {
  this.vehicle = vehicleProvider.get();
}
```

(モジュールのコードをまったく変更する必要がなかったことに注目してください。) このように書き直しても特に意味はありませんが、これを見ると、依存関係を直接要求する代わりにいつでも `Provider` を要求できることがわかると思います。

## スコープ

Guice はデフォルトで、要求された依存関係すべてに対して新しいインスタンスを作成します。オブジェクトが軽量な場合には、このポリシーは適切です。しかし作成が面倒な依存関係の場合には、いくつかのクライアントの間でインスタンスを共有する必要があります。リスト 19 では、`HeroModule` が `HeavyWaterRefinery` をシングルトンとしてバインドしています。

### リスト 19. シングルトンとしてバインドされた `HeavyWaterRefinery`

```
public class HeroModule implements Module {
  public void configure(Binder binder) {
    //...
    binder.bind(FuelSource.class)
      .to(HeavyWaterRefinery.class).in(Scopes.SINGLETON);
  }
}
```

#### シングルトンは悪者なのか

Web で「singletons are evil (シングルトンは悪者である)」を検索すると、シングルトンに関する大量のブログ投稿や記事、そしてシングルトンに対する不満をぶちまけた内容のものが見つかります。実は「アプリケーションのシングルトン」と (コメンテーターが抗議する種類の)「JVM のシングルトン」とは異なるのです。JVM のシングルトンは言語上のトリックを使うことで、またクライアントがシングルトンを参照する場合には静的参照を使うように強制することで「シングルトンらしさ」を強制します。一方アプリケーションのシングルトンでは、そうしたポリシーは強制されません。アプリケーションごとにクラスの 1 つのインスタンスが存在することが、(Guice などの) 別のレイヤーによって強制されるのです。クライアント・コードはこのルールを知る必要はなく、そのため設計が柔軟になり、テストを作成しやすくなります。

これはつまり、Guice が `HeavyWaterRefinery` を保持し、別のインスタンスで fuel source が必要になった場合には、Guice が同じ `HeavyWaterRefinery` を注入するということです。これによってアプリケーションの中で複数の `HeavyWaterRefinery` が立ち上げられる事態を避けることができます。

Guice ではスコープの選択に関して複数の方法が用意されています。つまりバインダーを使ってスコープを構成できる他、依存関係に直接アノテーションを付けることもできるのです (リスト 20)。

### リスト 20. バインダーの代わりにアノテーションを使ってスコープを選択する

```
@Singleton
public class HeavyWaterRefinery implements FuelSource {...}
```

Guice では `Singleton` スコープが始めから用意されていますが、必要な場合には独自のスコープも定義できるようになっています。その一例として、Guice のサーブレット・パッケージには `Request` と `Session` というに 2 つのスコープが別に用意されており、サーブレット・リクエストやサーブレット・セッションごとに、そのクラスに特有のインスタンスを提供することができます。

## 定数のバインディングとモジュールの構成

`HeavyWaterRefinery` を起動するためにはライセンス・キーが必要です。Guice では、新しいインスタンスをバインドするのと同じように定数値をバインドすることもできるのです。リスト 21 を見てください。

### リスト 21. モジュールの中で定数をバインドする

```
public class HeavyWaterRefinery implements FuelSource {
  @Inject
  public HeavyWaterRefinery(@Named("LicenseKey") String key) {...}
}

// in HeroModule:
binder.bind(String.class)
  .annotatedWith(Names.named("LicenseKey")).toInstance("QWERTY");
```

ここではバインディング・アノテーションが必要です。なぜならバインディング・アノテーションがないと Guice は `String` 同士を区別することができないからです。

ここで、先ほどは推奨しなかったにもかかわらず `@Named` アノテーションを使っていることに注目してください。その理由はリスト 22 のコードを示したかったからです。

### リスト 22. プロパティー・ファイルを使ってモジュールを構成する

```
//In HeroModule:
private void loadProperties(Binder binder) {
  InputStream stream =
    HeroModule.class.getResourceAsStream("/app.properties");
  Properties appProperties = new Properties();
  try {
    appProperties.load(stream);
    Names.bindProperties(binder, appProperties);
  } catch (IOException e) {
    // This is the preferred way to tell Guice something went wrong
    binder.addError(e);
  }
}

//In the file app.properties:
LicenseKey=QWERTY1234
```

このコードは Guice の `Names.bindProperties` ユーティリティー関数を使うことによって、app.properties ファイルの中にあるすべてのプロパティーを適切な `@Named` アノテーションを

持つ定数にバインドしています。これ自体が優れた方法ですが、これを見るとモジュールのコードをいくらでも複雑にできることがわかります。必要な場合には、データベースや XML ファイルからバインディング情報をロードすることもできます。モジュールは単純な Java コードであるため、非常に柔軟です。

## まとめ

Guice の主なコンセプトは次のように要約することができます。

- `@Inject` を使って依存関係を要求することができます。
- `Module` の中で実装に依存関係をバインドすることができます。
- `Injector` を使ってブートストラップ動作をさせることができます。
- `@Provides` メソッドを使うことで柔軟性を高めることができます。

Guice について知るべきことは他にもたくさんありますが、この記事で説明した話題を理解できれば Guice を使うことはできるはずです。Guice をダウンロードし、またこの記事のサンプル・コードもダウンロードして、さまざまなことを試してみるようにお勧めします。あるいはもっと良い方法として、独自のサンプル・アプリケーションを作成してみてください。本番用のコードを気にせずに Guice のコンセプトを試してみるのは非常に楽しいものです。Guice の高度な機能 (例えばアスペクト指向プログラミングのサポートなど) について学びたい場合には、「参考文献」に挙げたリンクを調べてください。

本番用のコードに関して言うと、DI の欠点の 1 つは伝染性があることです。1 つのクラスに注入すると、次から次へと注入することになりがちです。これは良いことかもしれません。DI によってコードが良いものになるからです。しかしその一方、既存のコードを大量にリファクタリングする羽目になるかもしれません。作業を管理可能なレベルに保つためには、Guice の `Injector` をどこかに保存し、それを直接呼び出すようにすることです。`Injector` は松葉杖のようなものです。必要なものではあっても、長期的には必要ないものなのです。

Guice 2.0 はまもなく登場するはずです。この記事では一部の機能を説明しませんでしたが、それらを使うとモジュールの構成が簡単になり、また大規模で高度な構成方式をサポートできるようになります。Guice 2.0 の機能について知るためには「参考文献」のリンク先を見てください。

皆さんがツールキットの中に Guice を追加するようになることを祈っています。私の経験では、DI はコードベースを柔軟でテストしやすくするには必須のものです。Guice によって DI が容易になり、さらには楽しいものにさえなります。作成が楽しい上に柔軟でテストしやすいコード以上に良いものがあるでしょうか。

---

# ダウンロード

| 内容 | ファイル名 | サイズ |
| --- | --- | --- |
| Java files for this article | j-guice.zip | 19KB |

# 著者について

Nicholas Lesiecki

Nicholas Lesiecki は 2000年以来、より良いソフトウェアについて執筆し、講演し、そして過度なほど考えています。彼はのんびりしているとき以外はシアトルにある Google のエンジニアリング・オフィスでコードを作成しています。