Persistente Python-Objekte mit der ZODB

Author: Christian Theune <ct@gocept.com>

Dies ist die Begleitdokumentation zum gleichnamigen Workshop auf dem Chemnitzer Linuxtag 2010. Die grundlegenden Themen werden im Detail wiedergegeben. Fortgeschrittene Themen werden nur angerissen, mit Hinweisen, wie interessierte Entwickler selbständig mehr herausfinden können.

Contents

Einleitung

Objektorientierte Anwendungen (wie man sie in Python schreiben kann) erzeugen zur Laufzeit viele Objekte. Diese Objekte und die dazugehörigen Daten gehen verloren, sobald der zur Anwendung gehörige Betriebssystemprozess beendet ist.

Um den Zustand eines Objekts über die Lebenszeit des Prozesses, der das Objekt erstellt hat, zu erhalten, benötigen wir eine Datenbank, derer es reichlich gibt. Der allgemein übliche Ansatz besteht darin den Zustand eines Objektes auszulesen und auf eine relationale Datenbank abzubilden. Dieses Vorgehen wird objekt-relationales Mapping genannt ("object-relational mapping", kurz ORM): aus einer Menge von Klassen wird eine Abbildung erzeugt, die erklärt, wie die Attribute von Objekten und Referenzen zwischen Objekten in einer relationalen Datenbank gespeichert werden kann - und umgekehrt.

Solche Abbildungen händisch zu definieren ist allerdings sehr aufwändig und fehleranfällig. Daher haben sich eine Reihe von Werkzeugen ("object-relational mappers") etabliert, die diesen Vorgang automatisieren und inzwischen auch sehr gut geworden sind.

Allerdings können selbst die besten Werkzeuge wie SQLAlchemy den Konzeptbruch ("impedance mismatch") der sich zwischen der objektorientierten und der relationalen Welt auftut nur teilweise schließen: einige Konstrukte der Objektorientierung sind nicht vollständig auf das relationale Modell abbildbar. Darunter fallen:

- beliebig-typisierte Verweise
- nicht-tabellenförmige (z.B. Baumartige) Daten
- variable Schemata von Objekten gleichen Typs

(Echte) Objektdatenbanken haben diese Beschränkungen nicht und kommen ohne eine Abbildung zwischen Objekten und ihrer Speicherform aus und bieten gleichzeitig grundlegende Datenbankeigenschaften. Obwohl relationale Datenbanken einfacher zu programmieren sind, sollte man sich jedoch einiger Einschränkungen bewusst sein:

- ODBMS sind langsamer bei der Verarbeitung großer Tabellenförmiger Daten als RDBMS
- die Datenbankstruktur ist an die Code-Struktur gebunden

Die ZODB ist solch eine echte Objektdatenbank für Python. Der Name bedeutet "Zope object database" -- die Datenbank wurde ursprünglich für Zope entwickelt, ist aber auch einzeln nutzbar.

Beispiel

Wir fangen mit einem Beispiel an, das die grundlegende Mechanik demonstriert, wir man eine Anwendung auf Basis der ZODB entwickelt.

Zuerst brauchen wir eine Umgebung, in der die ZODB installiert ist. Da die ZODB als Egg auf PyPI verfügbar ist, bieten sich verschiedene Wege an: *easy_install*, *zc.buildout* oder *virtualenv* können je nach Bedarf verwendet werden.

Der einfache Aufruf von *easy_install* sieht so aus:

```
$ easy_install ZODB3
```

> Hinweis: Der Name des Eggs ist aus historischen Gründen "ZODB3", obwohl der Name des Python-Pakets "ZODB" ist.

Mit dem interaktiven Python-Interpreter können wir die ZODB jetzt sofort benutzen. Wir erzeugen eine Datenbank, öffnen eine Verbindung und holen uns eine Referenz zum Wurzelobjekt:

```
$ python
>>> import ZODB
>>> db = ZODB.DB('Data.fs')
>>> conn = db.open()
>>> root = conn.root()
>>> root
{}
```

Wir sehen, dass das Wurzelobjekt ein Dictionary ist und können sofort damit anfangen Informationen darin zu speichern:

```
>>> root['breakfast'] = 'Spam, eggs, spam, spam, bacon, and spam.'
>>> root['breakfast']
'Spam, eggs, spam, spam, bacon, and spam.'
```

Die ZODB ist vollständig transaktional: daher müssen wir die laufende Transaktion zuerst comitten, um unsere Änderungen dauerhaft zu speichern:

```
>>> import transaction
>>> transaction.commit()
```

Wenn wir jetzt den Interpreter verlassen und neu starten können wir feststellen, dass unsere Änderungen gespeichert wurden:

```
>>> ^D
$ python
>>> import ZODB
>>> db = ZODB.DB('Data.fs')
>>> conn = db.open()
>>> root = conn.root()
>>> root
{'breakfast': 'Spam, eggs, spam, spam, bacon, and spam.'}
```

Transaktionen können auch abgebrochen werden. Dadurch werden alle noch offenen Änderungen an Objekten der Datenbank rückgängig gemacht:

```
>>> root['breakfast'] = 'Continental'
>>> root
{'breakfast': 'Continental'}
>>> transaction.abort()
>>> root
{'breakfast': 'Spam, eggs, spam, spam, bacon, and spam.'}
```

Wird der Python-Interpreter verlassen währen noch eine Transaktion läuft, wird dies als impliziter Transaktionsabbruch behandelt.

Wir haben jetzt gesehen, wie einfache Daten in einem Dictionary gespeichert wurden. Man kann allerdings auch komplexe Objekte (beinah alle Python-Objekte) in der ZODB speichern, solange diese "picklebar" sind. Betrachten wir folgende Klasse, die ein Bankkonto modelliert:

```
import persistent

class Account(persistent.Persistent):

    def __init__(self):
        self.balance = 0.0

    def deposit(self, amount):
        assert amount > 0
        self.balance += amount

    def cash(self, amount):
        assert amount < self.balance
        self.balance -= amount
```

Hinweis: Die Verwendung von `Persistent` als Basisklasse ist notwendig, damit die ZODB Änderungen an Attributen automatisch merkt und entsprechend reagieren kann. Objekte müssen aber nicht von dieser Klasse abgeleitet sein, um in der Datenbank gespeichert zu werden -- es erleichtert das Leben aber wesentlich. Mehr dazu betrachten wir im Abschnitt "Anwendungsentwicklung".

Wir können jetzt ein Exemplar dieser Klasse erzeugen und in der Datenbank speichern, indem wir es vom Wurzelobjekt aus referenzieren:

```
>>> root['knight-foundation'] = Account()
>>> root['knight-foundation'].balance = 10**10
>>> transaction.commit()
>>> ^D
$ python
>>> ... initialization code ...
>>> root['knight-foundation'].balance
10000000000L
```

Hiermit beschließen wir den Überblick über das grundlegende Vorgehen wie man Objekte in der ZODB speichert. Wir haben gesehen, wie man:

- eine Verbindung zur Datenbank aufbaut

- das Wurzelobjekt erhält und modifiziert
- Transaktionen comittet und abortet
- Objekte eigener Klassen speichert

Architektur

Die Architektur der ZODB besteht aus wenigen Elementen, die sich in zwei Gruppen unterteilen:

- Datenbankarchitektur
- Transaktionskoordination

Betrachten wir den Quelltext, den wir zur Initialisierung der Datenbank benutzt haben, lassen sich die verschiedenen Elemente bereits unterscheiden:

```
import ZODB.FileStorage.FileStorage
import ZODB.DB
import transaction

storage = ZODB.FileStorage.FileStorage('Data.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root()
```

> Hinweis: Die Initialisierung in unserem Eingangsbeispiel wurde in einer Kurzform geschrieben, wogegen unser Beispiel hier die vollständige Mechanik aus Sicht der Architektur verwendet.

In diesem Beispiel werden 4 Objekte erzeugt und 3 Module importiert.

Die Datenbankarchitektur berührt die drei Elemente **Wurzelobjekt** ("root"), **Datenbank** und **Storage** ("Speicher").

Das **Wurzelobjekt** ist der Einstiegspunkt in die Datenbank. Es existiert immer und kann nicht gelöscht werden. Alle anderen Objekte, die in der Datenbank gespeichert sind, sind von der Wurzel erreichbar, indem man sie traversiert (durch das Zugreifen auf Attribute, Indizes oder Funktionsaufrufe). Der so beschriebene Graph mit der Wurzel als wohlbestimmten Element wird als "Objektgraph" bezeichnet.

Das **Datenbankobjekt** implementiert die logische Datenbankschicht. Es verwaltet Verbindungen und kann Objekte aus der physischen Schicht laden und in die physische Schicht speichern.

Der **Storage** implementiert die physische Datenbankschicht und ist verwantwortlich dafür die serialisierte, flache Form von Objekten auf einem physischen Medium zu speichern.

Storages sind als Plugins verfügbar und für verschiedene Anwendungsfälle entwickelt und optimiert: FileStorage ist der "Gold"-Standard für einen robusten und skalierbaren Storage, der alle Features der ZODB unterstützt. ZEO implementiert eine Netzwerkschicht um Storages für mehrere Anwendungen gleichzeitig zugänglich zu machen, MappingStorage bietet eine reine hauptspeicherbasierte Datenbank, ...

Die Transaktionskoordination besteht aus zwei Elementen: Transaktionen und Verbindungen.

Transaktionen definieren den Umfang der Änderungen. Sie können mehrere Änderungen an mehreren Datenbanken zusammenfassen und garantieren ACID-kompatibles Verhalten. Die Transaktionskoordination der ZODB steht auch als separates Paket zur Verfügung und implementiert ein allgemein verwendbares Zwei-Phasen-Commit-Protokoll ("two phase commit", TPC) durch das die ZODB mit jeder anderen Datenbank, die dieses Protokoll unterstützt integriert werden kann. Vorgefertigte Integrationen für relationale Datenbanken (inkl. SQLALchemy) bieten Anwendungen die Möglichkeit relationale Datenbanken Seite an Seite mit der ZODB zu benutzen ohne auf die Sicherheit von Transaktionen verzichten zu müssen.

Wir haben gesehen, dass das Wurzelobjekt nicht direkt von der Datenbank gelesen werden kann: wir haben zuerst eine **Verbindung** geöffnet, die wiederrum in der Lage ist, Objekte aus der Datenbank zu laden und mit einer Transaktion verknüpft ist. Verbindungen erlauben auch, dass eine Anwendung mehrere Transaktionen parallel im gleichen Prozess verwenden kann und unterstützt thread-basierte und manuelle Kontextbestimmung.

Funktionsübersicht

Da wir nicht alle Funktionen der ZODB in diesem Tutorial abdecken können, hier eine (unvollständige, nicht sortierte) Liste von Funktionen, die für Entwickler interessant sein könnten:

- Konfigurierbare Storages
- Speichern fast beliebiger Python-Objekte
- Optimierte Typen für spezielle Anforderungen (BTrees, Blobs)
- Volle ACID-Kompatibilität
- Unabhängige "two phase commit"-Implementation um mehrere Datenbanken (auch anderer Hersteller) zu integrieren
- Zwischenspeichern innerhalb von Transaktionen (Savepoints)
- Undo / History / Time travel
- Anwendungskontrollierte Konfliktlösung
- Cache-Tuning um IO/CPU/Speicher-Bedarf zu optimieren
- Konfiguration von ZODB-Eigenschaften durch Administratoren

Anwendungsentwicklung

Konfiguration

Um die ZODB für eine Anwendung zu konfigurieren müssen ein Storage, dessen Parameter und die allgemeinen Parameter der Datenbank angegeben werden.

Den kürzesten Weg, haben wir bereits im Eingangsbeispiel gesehen, in dem eine ZODB mit dem FileStorage geöffnet wird und alle Parameter auf ihre Standardwerten gesetzt sind:

```
>>> import ZODB.DB
>>> db = ZODB.DB('Data.fs')
```

Man kann den Storage auch vorab initialisieren und dann der Datenbank übergeben. Ein MappingStorage würde wie folgt konfiguriert werden:

```
>>> import ZODB.MappingStorage
>>> storage = ZODB.MappingStorage()
>>> db = ZODB.DB(storage)
```

Optionen werden den Konstruktoren der Storages und/oder Datenbank mitgegeben. Das folgende Beispiel stellt die Anzahl der im Hauptspeicher gecachten Objekte mit Hilfe des Parameters `cache_size` ein:

```
>>> db = ZODB.DB('Data.fs', cache_size=1000)
```

Ein zweiter Weg, eine Datenbank zu konfigurieren besteht darin, die Konfiguration aus einer Konfigurationsdatei auszulesen. Damit überträgt man dem Administrator einer Anwendung die Möglichkeit (und Verantwortung) einen geeigneten Storage auszuwählen und die Datenbank entsprechend der eigenen Anforderungen zu konfigurieren. Die API ist ein einfacher Funktionsaufruf:

```
>>> import ZODB.config
>>> db = ZODB.config.databaseFromURL('zodb.conf')
```

Die Konfigurationsdateien haben eine Apache-ähnliche Syntax:

```
<zodb>
    <filestorage>
        path Data.fs
    </filestorage>
    cache-size 1000
</zodb>
```

Die Konfiguration kann auch mithilfe anderer Funktionen geladen werden, die erlaubt die Konfiguration aus anderen Quellen (z.B. URLs) zu laden oder stat einer vollen Datenbank nur einen Storage zu definieren.

Details zu den genauen Parametern, die in den Konfigurationsdateien erlaubt sind können der Datei `ZODB/component.xml` entnommen werden.

Die Konfiguration wird durch ein Paket namens `ZConfig` implementiert, welches seinerseits wiederrum erweiterbar ist, so dass Zusatzmodule (wie Storages) auch neue Konfigurationsoptionen definieren können, die sich nahtlos in die bestehenden ZODB-Konfigurationen integrieren.

Persistente Objekte

Die ZODB kann (beinah) alle Objekte speichern, ohne dass Anwendungsentwickler besondere Vorkehrungen treffen müssen. Objekte, die ihren Zustand verändern, erfordern jedoch, dass der ZODB solche Änderungen mitgeteilt werden. Andernfalls werden die Änderungen beim comitten einer Transaktion nicht dauerhaft gespeichert.

Betrachten wir eine Klasse:

```
class Followers(object):

    def __init__(self):
        self.followers = 0
```

Die unten stehende Datenbank-Interaktion illustriert das Problem:

```
>>> # Schritt 1: Das Objekt wird erzeugt und gespeichert.
>>> root['theuni'] = Followers()
>>> root['theuni'].followers
0

>>> # Schritt 2: Die sofortige Änderung wird gespeichert.
>>> root['theuni'].followers += 1
>>> root['theuni'].followers
1
>>> transaction.commit()
>>> root['theuni'].followers
1
>>> ^D
$ python
>>> ... initialization code ...
>>> root['theuni'].followers
1

>>> # Schritt 3: Folgende Änderungen werden nicht gespeichert.
>>> root['theuni'].followers += 1
>>> root['theuni'].followers
2
>>> transaction.commit()
>>> ^D
$ python
>>> ... initialization code ...
>>> root['theuni'].followers
1
```

Wir sehen, dass das Objekt in der ZODB gespeichert werden **kann** -- nur Änderungen an diesem Objekt werden später nicht mehr automatisch erfasst.

An dieser Stelle kommt die `Persistent`-Basisklasse wieder ins Spiel.

Diese Klasse trägt zwei Verantwortungen: automatisches Erfassen von Änderungen an Attributen von persistenten Objekten, sowie das Aufteilen von Objekten in Datenbanksätze. Von jetzt an nennen wir Objekte "persistent", wenn sie Exemplare der Klasse `Persistent` sind.

Das automatische Erfassen von Änderungen arbeitet direkt auf persistenten Objekten: wenn wir die `Followers`-Klasse von `Persistent` ableiten wird unser obiges Beispiel sich wie erwartet verhalten.

Änderungen werden **nicht** automatisch für nicht-persistente Unterobjekte erfasst: selbst wenn wir `Followers` von `Persistent` ableiten können wir kein `StringIO`-Objekt referenzieren und erwarten, dass Änderungen am `StringIO`-Objekt automatisch erfasst werden.

Hinweis: Angenommen, das `StringIO`-Beispiel würde Änderungen automatische erfassenm dann gäbe es keine Notwendigkeit überhaupt ein Objekt von `Persistent` abzuleiten, da das Wurzelobjekt bereits ein persistentes Dictionary ist und alle anderen Objekte von dort referenziert werden. `Persistent` verhält sich nicht transitiv!

Da Listen und Dictionaries veränderbare Objekte sind und sehr häufig gebraucht werden, gibt es im `persistent`-Modul bereits alternative Implementationenn für diese Typen, die sich im Hinblick auf Persistenz bereits korrekt verhalten und kompatibel zu den vorgefertigten Python-Typen sind:

```
>>> from persistent.list import PersistentList
>>> root['l'] = PersistentList()

>>> from persistent.list import PersistentDict
>>> root['d'] = PersistentDict()
```

Die zweite Verantwortung von `Persistent` liegt darin, Objekte in getrennte Datensätze der ZODB aufzutrennen.

Da der Serialisierungsmechanismus der ZODB `Pickle` ist, müssen wir vermeiden, dass das Wurzelobjekt mit all seinen referenzierten Objekten in einem einzigen großen String, der die ganze Datenbank enthält, gespeichert wird: wir müssten die ganze Datenbank sowohl in den Speicher laden, als auch den Pickle deserialisieren um ein einzelnes Objekt zu laden. Alle persistenten Objekte sorgen automatisch dafür, dass sie aus dem Pickle des referenzierenden Objektes entfernt werden (und nur einen Verweis hinterlassen) und erhalten einen eigenen Datensatz und eine Identität.

Die Objektidentität (OID) eines persistenten Objekts kann durch das `_p_oid`-Attribute ausgelesen werden:

```
>>> # Die Wurzel hat immer OID 0
>>> root._p_oid
'\x00\x00\x00\x00\x00\x00\x00\x00'
>>> root['knight-foundation']._p_oid
'\x00\x00\x00\x00\x00\x00\x00\x01'
```

Normalerweise ist eine OID ein 8 Byte langer Binärstring, der einen 64-Bit-Counter repräsentiert. Es ist nicht garantiert, dass alle OIDs lückenlos verwendet werden, da garantiert sein muss, dass OIDs innerhalb einer Datenbank eindeutig sein müssen und konfliktfrei erzeugt werden können müssen.

Daher müssen, wenn ein persistentes Objekt geändert wird, nur der Pickle des betroffenen Objekts sowie der nicht-persistenten Unterobjekte gespeichert werden, was Zeit beim Picklen und Schreiben auf die Festplatte spart.

Zusätzlich zum besseren Schreibverhalten, wirkt es sich auch positiv auf das Leseverhalten aus, da nur die persistenten Objekte, die tatsächlich berührt werden, aus der Datenbank geladen werden müssen.

Nicht-picklebare Objekte

Fast alle, aber eben nicht alle, Objekte lassen sich in der ZODB speichern.

Ein Objekt, das in der ZODB gespeichert werden soll, muss zuallererst picklebar sein. Typische, nicht-picklebare Objekte sind beispielsweise Sockets und

Dateien. Allgemein gesprochen all solche Objekte, die den Zustand externer Resourcen wiederspiegeln und deren Bedeutung sich nicht serialisieren und auf ein anderes System übertragen lässt.

Da große Mengen an binären Daten sich am besten in Dateien speichern lassen hält die ZODB einen speziellen Datentyp bereit, um diese Art von Information zu speichern: Blobs.

Blobs sind Objekte, die eine `open`-Methode haben, die sich fast genauso wie Python's builtin-Funktion zum Öffnen von Dateien verhält:

```
>>> from ZODB.blob import Blob
>>> root['data'] = Blob()
>>> f = root['data'].open('w')
<open file '/tmp/tmpWfKa18', mode 'wb' at 0x9f0cd3c>
>>> f.write('Spam!')
```

Der Aufruf von `open` gibt eine geöffnete Datei zurück, mit der man wie mit allen anderen Dateien umgehen kann: lesen, schreiben, suchen, schliessen ...

> Hinweis: Nicht alle Storages unterstützen die Blob-API (FileStorage und ZEO unterstützen sie auf jeden Fall), aber es gibt einen allgemeinen "Wrapper", der andere Storages um die Blob-Funktionalität erweitern kann. Dieser Wrapper heißt `BlobStorage`.

Um Blobs mit einer Datenbank benutzen zu können muss der Storage konfiguriert werden ein entsprechendes Blob-Verzeichnis zu benutzen, in dem entsprechend große Mengen an Daten gespeichert werden können.

Wird diese Konfiguration nicht vorgenommen, gibt es einen entsprechenden Fehler bei der Verwendung von Blobs:

```
>>> transaction.commit()
Traceback (most recent call last):
...
Unsupported: Storing Blobs in <ZODB...FileStorage ...> is not supported.
```

Der Konstruktor von `FileStorage` unterstützt ein entsprechendes Argument, um das Blob-Verzeichnis einzustellen:

```
>>> fs = ZODB.FileStorage.FileStorage('Data.fs', blob_dir='my-blobs')
```

Blobs sind in Wirklichkeit nur ganz normale Dateien, die von der ZODB verwaltet werden, und brauchen daher ein wenig mehr Aufmerksamkeit als andere Objekte in der ZODB. Solange zu einem Blob geöffnete Dateien existieren, können wir eine Transaktion nicht committen:

```
>>> transaction.commit()
Traceback (most recent call last):
...
ValueError: Can't commit with opened blobs.
```

Sobald wir die Datei schliessen, können wir die Transaktion auch comitten:

```
>>> f.close()
>>> transaction.commit()
```

Ansonsten unterstützen Blobs jedoch alle Funktionen die die ZODB anbietet: Transaktionsabbruch, Savepoints, Undo, History, konfigurierbare Storages, ZEO, ...

Transaktionen

Transaktionen ermöglichen es Änderungen an einer Datenbank in kontrollierter Art und Weise vorzunehmen. "Kontrolliert" bedeutet in diesem Zusammenhang üblicherweise, dass die Anforderungen des "ACID"-Prinzips erfüllt werden, welches wiederrum einzelne Eigenschaften beschreibt, die für Datenbanken erwünscht sind. Diese Eigenschaften sind nur informell beschrieben (die entsprechenden Wikipedia-Einträge sind recht gut) und lassen sich in etwa wie folgt zusammenfassen:

Atomare Änderungen (Atomicity) Entweder alle Änderungen einer Transaktion werden permanent auf die Datenbank angewendet, oder gar keine.

Konsistenz (Consistency) Wenn die Datenbank vor Beginn der Transaktion in einem konsistenten Zustand war, dann wir die Datenbank nach Ende der Transaktion ebenfalls in einem konsistenten Zustand sein. Konsistenz bezieht sich in diesem Zusammenhang auf Schlüssel und Fremdschlüssel, sowie explizite Bedingungen (Constraints), die zu jedem Zeitpunkt erfüllt sein sollen.

Isolation Erfordert, dass zwei parallel ausgeführte Transaktionen sich nicht gegenseitig beeinflussen.

Dauerhaftigkeit (Durability) Wenn eine Änderung von der Datenbank committet wurde, muss diese Änderung garantiert dauerhaft sein, insbesondere in Hinblick auf Systemabstürze.

Die ZODB implementiert diese Eigenschaften sehr gründlich, wobei nur einige prinzipiell schwierige Ausnahmefälle bestehen. Die Implementation basiert auf einer MVCC-Strategie (multi-version concurrency control) die parallele Lese- und Schreibzugriffe erlaubt ohne auf Locking angewiesen zu sein und Isolationsprobleme vermeidet. Im Detail bedeutet dies, dass alle Transaktionen eine konsistente Sicht auf die Datenbank erhalten, wie sie zum Zeitpunkt des Transaktionsbeginns bestanden hat und Änderungen von Transaktionen, die später begonnen aber schneller verarbeitet wurden, nicht sehen.

Eine einfache Programmierschnittstelle erlaubt es den Lebenszyklus von Transaktionen zu verwalten.

transaction.begin()

ZODB-Verbindungen sind immer mit einer Transaktion assoziiert, so dass Anwendungsentwickler nicht gezwungen sind eine Transaktion explizit zu beginnen.

Die API stellt trotzdem eine Funktion zur Verfügung um eine Transaktion explizit zu beginnen. Diese wird typischerweise aufgerufen kurz bevor eine Anwendung Daten liest, Berechnungen anstellt und diese in der Datenbank speichert. Dieses Vorgehen stellt sicher, dass die Anwendung einen Datenbankstand sieht, der so aktuell wie möglich ist. Ein Transaktionsbeginn bricht implizit die bereits bestehende Transaktion ab, so dass alle offenen Änderungen rückgängig gemacht werden.

Dieses Vorgehen ist bei Web-Servern zum Beispiel wertvoll, da Threads (oder Prozesse) eventuell seit geraumer Zeit keine Anfragen verarbeitet haben und daher mit einer sehr alten Sicht auf die Datenbank arbeiten würden.

Die Verwendung von `begin` ist ein simpler Funktionsaufruf:

```
>>> transaction.begin()
```

transaction.commit()

Um Änderungen einer Transaktion dauerhaft zu speichern, muss die Transaktion "comittet" werden. Der dazugehörige Funktionsaufruf ist simpel:

```
>>> transaction.commit()
```

Falls eine Transaktion nicht erfolgreich comittet werden kann (aufgrund von Konsistenzproblemen, Isolationsproblemen, Ressourcenproblemen, ...) wird eine Exception geworfen:

```
>>> transaction.commit()
Traceback (most recent call last):
...
ConflictError: ...
```

Sobald ein Fehler während eines Commits aufgetreten ist, wird die Transaktion in einen "nicht comittbaren" Zustand versetzt: der Zustand der Transaktion lässt sich noch auslesen und "zur Seite legen", aber die Transaktion muss schlussendlich abgebrochen und neu begonnen werden um mit der Verbindung weiter zu arbeiten:

```
>>> transaction.commit()
Traceback (most recent call last):
...
TransactionFailedError: An operation previously failed, with traceback:
...
>>> transaction.abort()
```

transaction.abort()

Anstatt eine Transaktion zu committen, kann sie auch abgebrochen werden, was alle Änderungen rückgängig macht:

```
>>> transaction.abort()
```

> Hinweis: Es sind nur solche Änderungen betroffen, die an Objekten in der Datenbank gemacht wurden. Objekte, die nicht zur Datenbank gehören werden nicht zurückgesetzt.

transaction.savepoint()

Transaktionen sind, per definition, atomar.

In einigen Situationen (wie Stapelverarbeitung) kann es jedoch sinnvoll sein, einzelne Teile des Stapels zu verarbeiten und fehlerhafte Teile auszusortieren ohne den gesamten Vorgang abzubrechen und damit aufwändige Berechnungen wiederholen zu müssen.

Hierbei helfen "Savepoints".

Savepoints werden erzeugt, indem man die Funktion `savepoint` aufruft, welche ein Savepoint-Objekt zurückliefert. Dieses Objekt ist ein Zeiger auf den Zustand der Transaktion, wie sie zum Zeitpunkt des Funktionsaufrufs war, und stellt die Methode `rollback` zur Verfügung, die benutzt werden kann, um die Transaktion wieder in den früheren Zustand zu versetzen, wenn gewollt:

```
>>> root['data'] = 1
>>> s1 = transaction.savepoint()
>>> root['data'] = 2
>>> s2 = transaction.savepoint()
>>> root['data'] = 3
>>> s1.rollback()
>>> root['data']
1
```

Es können auch mehrere Savepoints erzeugt und zurückgerollt werden. Allerdings verfällt die Gültigkeit neuerer Savepoints, sobald man zu einem früheren Savepoint zurückkehrt. Die Gültigkeit noch früherer Savepoints wird dadurch aber nicht beeinflusst.

Eine spezielle Variation von Savepoints bilden die sogenannten "optimistischen" Savepoints: man kann sie nicht zurückrollen, aber sie signalisieren der Datenbank, dass es ein guter Zeitpunkt wäre um Caches und Speichernutzung zu optimieren:

```
>>> transaction.savepoint(optimistic=True)
```

transaction.doom()

Die bisher genannten Funktionen zur Transaktionsverwaltung decken die Standardfunktionen von fast allen Transaktionsmanagern ab.

Die ZODB bietet eine zusätzliche Funktion, die zwar selten gebraucht wird, im Falle des Falles aber sehr nützlich ist: "verdammte" Transaktionen.

Eine Transaktion zu "verdammen" kann man sich auch als verzögerten Transaktionsabbruch vorstellen: wenn eine Transaktion verdammt wurde, bleibt ihr Zustand erhalten und wir können weitere Berechnungen vornehmen. Die ZODB wird jedoch (garantiert) verhindern, dass diese Transaktion zu keinem Zeitpunkt comittet werden kann -- sie kann nur noch abgebrochen werden:

```
>>> root['x'] = 1
>>> transaction.doom()
>>> root['x']
1
>>> transaction.commit()
Traceback (most recent call last):
...
DoomedTransaction
>>> transaction.abort()
```

Dieses Verhalten ist in Fehlerfällen nützlich, in denen es nicht sinnvoll ist die Transaktion zu comitten, der aktuelle Zusdtand aber noch benötigt wird um sinnvolle Fehlermeldungen auszugeben.

Testen von ZODB-Anwendungen

Software zu testen, die Daten in Datenbanken speichert erfordert meist viel Aufwand um

- Testumgebungen zu konfigurieren

- Den Zustand der Datenbank zu kontrollieren
- Commits zu vermeiden oder rückgängig zu machen
- "Mini"-Datenbanken zu unterstützen

Anwendungen auf Basis der ZODB lassen sich sehr einfach für Testzwecke anpassen. Dabei helfen zwei speziell entwickelte Storages: MappingStorage und DemoStorage.

MappingStorage wird verwendet um konfigurationsfreie, reine hauptspeicherbasierte Datenbanken zu erzeugen, die trotzdem alle Features (mit Ausnahme von Blobs) der ZODB unterstützen und keine Daten im Dateisystem ablegen. Dies ist eine wichtige Eigenschaft um zu verhindern, dass man aus Versehen mit Produktionsdatenbanken testet: MappingStorage kann nicht aus Versehen den FileStorage überschreiben. Dadurch kann man Tests auch parallel zur laufenden Anwendung auf einem Produktivsystem ausführen. Ausserdem: Testdatenbanken hinterlassen keine Spuren im Dateisystem -- auch bei abgebrochenen (oder abgestürzten) Testläufen.

Eine Datenbank auf Basis von MappingStorage wird wie folgt angelegt:

```
>>> import ZODB.MappingStorage
>>> ms = ZODB.MappingStorage()
>>> db = ZODB.DB(ms)
```

DemoStorage wird benutzt um bereits existierende Storages mit einem Wrapper zu versehen, der den bisherigen Stand des Storages wiedergibt, aber Schreibzugriffe und Transaktionen vom Backend isoliert:

```
>>> import ZODB.DemoStorage
>>> ds = ZODB.DemoStorage(ms)
>>> db = ZODB.DB(db)
```

Kombiniert man diese beiden Storages, so können wir Test-Umgebungen mit einer Hauptspeicherdatenbank erzeugen und, bevor ein Test ausgeführt wird, diese mit einem DemoStorage umwickeln. Sobald der einzelne Test ausgeführt wurde, wird der DemoStorage wieder entfernt und durch einen frischen DemoStorage ersetzt, bevor der nächste Test ausgeführt wird. Dadurch werden alle Tests optimal voneinander isoliert (zumindest aus Sicht der Datenbank) und wir vermeiden es teure Setup-Routinen je Test erneut auszuführen.

Ein anderes Szenario, in dem DemoStorage eine interessante Rolle spielt ist die Nutzung von Produktionsdatenbanken um neuen Code oder Migrationen auszuprobieren. Da Demostrage jeden beliebigen anderen Storage im Hintergrund verwenden kann (wie FileStorage oder ZEO) kann man vermeiden große Datenbanken häufig oder überhaupt zu kopieren.

Skalierbarkeit

Die Skalierbarkeit von Datenbanken lässt sich an zwei Dimensionen betrachten: Transaktionsrate sowie Speichern und Verarbeiten vieler Objekte. In diesem Abschnitt betrachten wir ausgewählte Probleme und wie diese mit Hilfe von ZODB-spezifischen Technologien gelöst werden können.

Transaktionsrate

Die Anzahl der Transaktionen die in einer begrenzten Zeit verarbeitet werden hängt von der CPU-Zeit und den IO-Operationen ab, die eine Anwendung benötigt.

Zope Enterprise Objects (ZEO)

In vielen Fällen ist die Performance der Anwendung hauptsächlich abhängig von der CPU-Zeit, die zur Verfügung steht. Im großen Stil lässt sich Rechenzeit heutzutage durch mehr Prozessorkerne (auf einer oder mehr Maschinen) realisieren. Dies bedeutet, die Rechenlast muss über mehrere Prozesse verteilt werden und sollte netzwerktransparent sein. (Dies behebt gleichzeitig auch das Problem von Python's GIL.)

Um eine ZODB von mehreren Prozessen nutzbar zu machen, benötigt man einen Storage, der ein entsprechendes Protokoll anbietet. Die generische Lösung ist bekannt als ZEO (Zope Enterprise Objects). ZEO bietet einen Serverprozess, der ein oder mehrere Storages per Netzwerk exportieren kann und eine Client-Implementation (ClientStorage) die in der ZODB-Anwendung zum Einsatz kommt. Damit kann eine prinzipiell beliebige Anzahl von Clients mit einer ZODB gleichzeitig arbeiten.

Praktisch gibt es einige Einschränkungen in Bezug auf die Anzahl von Clients, die ein einzelner ZEO-Server bewältigen kann. Diese lassen sich zum größten Teil, durch die sogenannte "Fan-out"-Konfiguration beseitigen, bei der mehrere ZEO-Server hierarchisch hintereinander geschaltet werden.

Der ZEO-Prozess selbst ist hauptsächlich durch die IO-Geschwindigkeit des angeschlossenen Storages beschränkt und dies wiederum hauptsächlich durch die Schreibgeschwindigkeit (Rate und Latenz) der physischen Medien. Nur selten ist ein ZEO-Server durch die CPU-Geschwindigkeit limitiert.

> Hinweis: Beispiele zur Konfiguration einer ZEO-Umgebung ist im Abschnitt "Wartung und Betrieb" zu finden.
>
> Hinweis: Spezifische Anwendungen, Lastprofile und Anforderungen erfordern spezifische Analysen für Optimierungen. Allgemeine Tipps sind nur selten hilfreich und richten eher Schaden an.

Caching

IO-Zugriffe belasten Festplatten und Netzwerk (ZEO). Um überflüssige Zugriffe zu reduzieren hat jede Verbindung der ZODB einen Cache um "lebendige" Objekte im Hauptspeicher zu halten und in den folgenden Transaktionen wiederverwenden zu können.

Da der Korpus einer normalen Datenbank meist weitaus größer als der verfügbare Hauptspeicher ist müssen diese Caches begrenzt werden. Da jede Verbindung außerdem ihren eigenen Cache hält muß man damit rechnen ein Vielfaches der erwarteten Größe im Speicher zu halten. Zope wird in der Standardkonfiguration beispielsweise bis zu 4 oder mehr Caches parallel halten, was auch bis zu 4 mal so viel Objekte im Speicher sorgen kann, als in der Datenbank existieren.

Objektcaches können nach zwei Kriterien begrenzt werden: Anzahl der Objekte, Speichernutzung des Caches.

Der klassische Ansatz ist die Begrenzung auf eine Zahl von Objekten. Der Standardwert hierfür beträgt wenige hunder Objekte und gibt damit ein sehr konservatives Verhalten mit niedrigem Speicherbedarf. Diese Form der Cachebegrenzung entfernt Objekte, die lange nicht mehr gebraucht wurden, bevorzugt aus dem Cache (LRU). Die Konfiguration erfolgt über den Parameter `cache_size`:

```
>>> db = DB(storage, cache_size=5000)
```

Ein neuer Ansatz ist ab ZODB 3.9 verfügbar und erlaubt es eine erwünschte Obergrenze für den Speicherverbrauch des Caches anzugeben. Dieser Cache benutzt die Größe des Pickles eines Objektes als Annäherung an den Speicherbedarf des Objekts. Diese Annäherung ist sehr grob und sollte nicht auf's Byte genau versetanden werden -- nicht einmal auf's Megabyte.

Der speicherbegrenzte Cache kann mit dem Parameter `cache_size_bytes` konfiguriert werden:

```
>>> db = DB(storage, cache_size_bytes=50*(1024**2))
```

Hinweis: Alle Begrenzungen können während einer Transaktion temporär überschritten werden, da Caches nur an Transaktionsgrenzen reduziert werden. Je nach Plattform kommt es außerdem noch darauf an, wie Python's genaues Speicherverhalten aussieht: eventuell wird freigegebener Speicher dem Betriebssystem wieder zur Verfügung gestellt oder auch nicht.

In Kombination mit ZEO gibt es noch eine weitere Cache-Form: den ZEO client cache. Dieser speichert Pickle-Daten, die vom ZEO-Server ausgeliefert wurden auf dem Client in einer Datei zwischen, deren Größe statisch angegeben werden kann. Dieser Cache gilt einmal für alle parallelen Verbindungen aus einem Prozess und minimiert die Effekte von Netzwerklatenz beim Lesen. Er ist üblicherweise deutlich größer als der Objektcache im Hauptspeicher, kann sogar gegebenenfalls die gesamte ZODB enthalten.

Konfliktfehler vermeiden

Eine weitere Quelle von verbrannter CPU-Zeit sind Konfliktfehler: diese Fehler treten auf, wenn zwei Transaktionen konkurrierend das gleiche Objekt ändernn wollten. Die ZODB wirft beim Commit der späteren Transaktion einen `ConflictError`.

Ein typischer Ansatz mit Konfliktfehlern umzugehen besteht darin, die fehlgeschlagene Transaktion abzubrechen und noch einmal von Neuem zu verarbeiten, in der Hoffnung, dass diesmal kein Konflikt auftreten wird.

Für Systeme, die bereits unter Last stehen kann dieses Verhalten dazu führen, dass die Last auf kritische Werte ansteigt, wenn Transaktionen plötzlich mehrfach ausgeführt werden müssen -- was wiederum dazu führen kann, dass noch mehr Transaktionen Konflikte aufweisen, was wiederrum ...

Dies ist ein inhärentes Problem mit allen Datenbanken, die ACID-Eigenschaften bereitstellen. Allerdings kann man mit etwas Vorsicht beim Programmieren viele typische Probleme vermeiden:

- "Schreiben beim Lesen" vermeiden: Anfragen, die keinen Schreibzugriff erfordern, sollten auch nicht (zufällig) einen auslösen

- Keine singulären Objekte erzeugen, die bei jeder Transaktion aktualisiert werden müssen. (Counter um Besucher zu zählen)
- Wenn hohe Schreibraten notwendig sind, sollte auf spezialisierte Datenstrukturen wie BTrees und Length-Objekte zurückgegriffen werden.
- Bei sehr hoher Schreiblast oder rechenintensiven Schreibzugriffen sollte man überlegen diese Berechnungen asynchron durchzuführen.
- In Ausnahmefällen kann "application conflict resolution" die richtige Antwort sein. Aber Vorsicht: die möglichen Performance-Steigerungen erkauft man sich mit dem teilweisen Verlust der ACID-Eigenschaften und sollte mit sehr viel Vorsicht passieren.

Umgang mit vielen Objekten: große Kollektionen

Ein Teil mit vielen Objekten umzugehen besteht darin effizient große Kollektionen zu verarbeiten.

Listen und Dictionaries (auch die persistenten) eignen sich intuitiv hervorragend um Daten zu speichern. Der Zugriff auf ein einzelnes Element dieser Strukturen erfordert allerdings, dass die gesamte Kollektion und gegebenenfalls auch der enthaltenen Werte in den Hauptspeicher geladen werden muss -- was viel IO und CPU-Zeit benötigt.

Die ZODB stellt eine BTree-Implememtation bereit, die beim Umgang mit großen Objektkollektionen hilft. Dies funktioniert nach dem Prinzip große Kollektionen in kleinere Teile zu zergliedern und dann in einem balancierten Baum anzuordnen. Jeder kleine Teil erhält einen eigenen Datensatz und kann damit separat geladen werden. In sehr großen Kollektionen müssen damit deutlich weniger Daten geladen werden um auf einige oder wenige Elemente einer Kollektion zuzugreifen.

Die BTree-Typen sind in C implementiert und nach den möglichen Typen der Schlüssel und Werte optimiert. Die möglichen Typoptimierungen sind: integer, long, float und object.

Die API von BTrees ist ähnlich (aber nicht identisch) der von Dictionaries, bietet aber auch Zähler und Mengen-Varianten:

```
>>> from BTrees.IOBTree import IOBTree
>>> root['b'] = IOBTree()
>>> len(root['b'])
0
>>> root['b'][1] = 'Foo'
>>> root['b'][2] = 'Bar'
>>> del root['b'][1]

>>> from BTrees.IOBTree import IOTreeSet
>>> root['s'] = IOTreeSet()
>>> root['s'].insert(10)
1
>>> root['s'].insert(11)
1
>>> root['s'].insert(10)
```

```
0
>>> 11 in root['s']
True
```

Zusätzlich zu den bekannten Operationen bieten BTrees die Möglichkeit den kleinsten und größten Schlüssel, sowie Untermengen anhand von Schlüsselbereichen effizient abzufragen:

```
>>> root['b'].maxKey()
2
>>> root['b'].minKey()
1
>>> list(root['b'].keys(min=2, max=1000))
[2]
```

BTrees verringern ausserdem das Risiko von Konfliktfehlern wenn mehrere Transaktionen den gleichen BTree bearbeiten, da Änderungen am BTree nur einzelne Teile bearbeiten. Sobald ein BTree hinreichend groß ist und die Zugriffsmuster weitgehend zufällig sind, dann liegen diese Teile häufig weit auseinander.

Umgang mit vielen Objekten: Indizierung und Abfrage

Ein anderer Aspekt im Umgang mit vielen Objekten besteht darin, einzelne Objekte schnell in der Datenbank ausfindig zu machen, ohne alle möglichen Objekte zu materialisieren.

Der Schlüssel zu effizienten Abfragen liegt darin Indizes zu erstellen, die die notwendige Information in einem Abfrage-orientierten Format speichert. Die weiter oben vorgestellten BTrees sind ein Hilfsmittel, dass bei der Erstellung von Strukturen auf der Anwendungsebene helfen kann um innerhalb der ZODB Indizierungslösungen zu erstellen.

Einige Pakete helfen bei der Erstellung solcher anwendungsspezifischer Lösungen: `zope.index` und `zope.catalog` nutzen BTrees um spezielle Indizes und ein auf Events-basierendes Gesamtsystem zu definieren. Mit diesem Ansatz erzeugt man im allgemeinen eine tabellarische Sicht auf den Objektgraphen, fast wie in einer relationalen Datenbank.

Indizes werden typischerweise nach spezifischen Abfrage oder Speicherkriterien optimiert und stehen für atomare Werte, Volltext, Mengenoperationen, Zeiträume und andere Szenarien bereits zur Verfügung.

Pflege und Betrieb

ZODB-Datenbanken sind einfach zu pflegen und erfordern nur wenig Aufmerksamkeit im täglichen Betrieb. Das Hauptaugenmerk liegt dabei auf der Vermeidung von Ressourcenüberlastung und Datensicherheit.

Installation von ZEO mit zc.buildout

Der Betrieb von ZEO erfordert die Installation einer separaten Komponente, zusätzlich zur eigentlichen Anwendung. Für `zc.buildout` gibt es vorgefertigte Rezepte um diese Konfiguration zu vereinfachen:

```
[buildout]
parts = zeo

[filestorage]
recipe = zc.recipe.filestorage
location = ${buildout:parts-directory}/database

[zeo]
recipe = zc.zodbrecipes:server
zeo.conf =
    <zeo>
        address 8100
        monitor-address 8101
        transaction-timeout 300
    </zeo>
    <filestorage 1>
        path ${filestorage:location}
        blob-dir ${database:location}/blobs
    </filestorage>
```

Eine ZODB-Anwendung kann sich nun als Client verbinden, indem sie folgende Konfiguration verwendet:

```
<zodb>
    <zeoclient>
        server 127.0.0.1:8100
        storage 1
        cache-size 500 MB
        blob-dir /tmp/blobs
    </zeoclient>
</zodb>
```

ZEO caches

Wie oben erwähnt benutzt ZEO einen Dateisystem-basierten Cache zur Reduzierung von Netzwerkzugriffen. Die Dateigrösse kann in der Konfiguration des Clients mit der Option `cache-size` eingestellt werden.

Da Blobs speziell verwaltete Dateien sind brauchen diese beim Einsatz mit ZEO ein separates Blob-Cache-Verzeichnis um die Blob dort temporär abzulegen. Ab ZODB 3.9 wird dieses Verzeichnis automatisch verwaltet und auf eine maximale Zielgrösse getrimmt (`blob-cache-size`, `blob-cache-size-check`). Abhängig von der Einstellung der Option `blob-cache-size-check` kann die Größe zu einem spezifischen Zeitpunkt jedoch deutlich größer als der in `blob-cache-size`) eingestellte Wert sein!

Alternativ kann der Blob-Cache jedoch auch auf Dateisystem-Ebene mit dem ZEO-Server geteilt werden, beispielsweise durch ein Netzwerkdateisystem. Dadurch müssen Blob-Daten nicht durch das ZEO-Protokoll übertragen werden, was den ZEO-Server entlastet und die Latenz verringert. In diesem Fall werden keine Blob-Daten lokal gespeichert und alle Blobs sind instantan und mit wahlfreiem Zugriff les- und schreibbar.

Backup

Das Sichern einer Datenbank hängt von den Eigenschaften des verwendeten Storage ab.

FileStorage unterstützt "hot snapshots", was bedeutet, dass die Datei "Data.fs" mit einfachen Dateioperationen kopiert werden können. (Die Dateien ".index" und ".tmp" brauchen keine Sicherung.)

Der Nachteil dieses Ansatzes besteht darin, dass bei jedem Backuplauf die **gesamte** ZODB gesichert wird und viele Backupsysteme große Dateien, die sich häufige ändern nicht mögen.

Um diesen Nachteil auszugleichen kann man das Werkzeug `repozo` verwenden. Dieses erzeugt kleine Backup-Dateien, die genau die neuen Transaktionen seit dem letzten Backup enthalten.

Packen

FileStorages haben die Eigenschaft nur zu wachsen, selbst wenn Objekte gelöscht werden, was neue Anwender häufig zunächst verwundert.

Da FileStorage nichts anderes als ein großes Log aller Transaktionen ist (um Undo und History zu unterstützen) muss regelmäßig ein Wartungszyklus durchgeführt werden, der nicht mehr benötigte Daten entfernt. Dieser Wartungszyklus heißt "Packen".

Packen kann man direkt aus einer ZODB-Anwendung, indem man die Funktion `pack` aufruft und angibt wieviele Tage von historischen Datensätzen aufgehoben werden sollen:

```
>>> db = ZODB.DB('Data.fs')
>>> db.pack(days=2)
```

Wenn man einen ZEO-Server betreibt, kann das Skript `zeopack` verwendet werden um außerhalb von einer Anwendung den Wartungszyklus für einen Storage anzustoßen:

```
$ zeopack -h localhost -p 8100 -d 2
```

Hinweis: Packen ist eine "online"-Operation auf der Datenbank, kann aber die Performance zeitweilig verringern.

Hinweis: Packen von FileStorages hält eine Kopie der Originaldatenbank vor dem Packen parat und benötigt freien Plattenplatz um die gepackte und die ungepackte Datenbank gleichzeitig zu speichern. Die Datei `Data.fs.old` kann nach Abschluß des Packlaufs gelöscht werden.

Übersicht weiterführender Themen

Sobald man mit der ZODB warm geworden ist, gibt es einige interessante Themen, die im Rahmen dieses Handouts etwas weit führen: eine Auflistung dieser Themen mit einen Erläuterungen soll als Ausblick dienen.

Schema-Änderungen

Da die Datenbank von der Struktur der Anwendungssoftware abhängt ist es gefährlich Klassen einfach umzubennenen oder in andere Module zu verlagern.

`zope.app.generation` setzt ein Muster um, mit dem man Software- und Datenbankaktualisierungen verlässlich und strukturiert miteinander verbinden kann. So ist sichergestellt, dass nach dem Start einer neuen Softwareversion die Datenbank automatisch auf die neuen Strukturen angepasst wurde.

Dieses Paket ist momentan relativ stark im ZTK verankert, soll aber noch entkoppelt werden um es auch für nicht-ZTK-Anwendungen zugänglich zu machen.

_p_changed-Magie um Nicht-persistente Objekte zu speichern

Objekte, die nicht von `Persistent` abgeleitet sind können zusammen mit persistenten Objekten gespeichert werden. Um Änderungen an diesen Objekten der Datenbank zu signalisieren, kann auf dem referenzierenden persistenten Objekt die Funktion `_p_changed` gerufen werden.

Aber: selten wird `_p_changed` richtig angewendet: Entwickler, die mit der ZODB erst seit kurzem Arbeiten, nutzen es zu häufig, was zu Flaschenhälsen (weil viele Lese-Operationen in Schreiboperationen enden) und übermäßigen Konfliktfehlern führt (weil immer die gleichen Objekte wieder und wieder geschrieben werden).

Daher: `_p_changed` existiert, es gelten aber die gleichen Regeln wie für Metaklassen: wenn man es verstanden hat, braucht man es eigentlich nicht mehr.

Undo

Undo erlaubt es frühere Transaktionen konsistent rückgängig zu machen und kann in einigen Situationen lebensrettend sein:

```
>>> db = ZODB.DB('Data.fs')
>>> tid = db.lastTransaction()
>>> db.undo(tid)
```

Dadurch wird eine neue Transaktion erzeugt, die die alte Transaktion rückgängig macht.

Um die Konsistenz zu sichern kann Undo keine Transaktionen rückgängig machen, die Objekte berühren welche in späteren Transaktionen schon geändert wurden. In diesem Fall kann es helfen, mehrere Transaktionen gemeinsam zurückzurollen.

> Hinweis: Undo ist im allgemeinen nicht für anwendungsspezfisches Verhalten geeignet, sondern eher ein Werkzeug für Administratoren und Debugging.

Mehrere Datenbanken

Die ZODB kann mehrere Datenbanken von verschiedenen Storages gleichzeitig verwalten und die Transaktionen synchronisieren. Objekte einer Datenbank können transparent auch Objekte anderer Datenbanken referenzieren.

Hinweis: Packen kann die Konsistenz dieser Referenzen stören, da Referenzen "von außen" währen der Garbage-Collection nicht berücksichtigt werden. Neuere Werkzeuge, die diesen Umstand abfedern wurden von Jim Fulton erarbeitet und erlauben Packen ohne Garbage-Collection bzw. können Garbage-Collection ausführen und dabei Referenzen aus externen Quellen berücksichtigen.

Erweitern und Entwickeln der ZODB

Die ZODB selbst muss nur selten selbst geändert werden um den eigenen Bedürfnissen angepasst zu werden. Jedoch gibt es Erweiterungspunkte, die es erlauben spezifische wiederkehrende Probleme individuell zu lösen. Dazu gehören Storages um Daten in abweichenden physischen Formaten zu speichern, sowie "data manager", die andere Datenbanken oder Systeme in das Transaktionsmanagement integrieren (wie z.B. E-Mail an einem günstigen Punkt innerhalb der Transaktion zu versenden).

Referenzen

Die ZODB-Homepage http://zodb.org

Das Subversion-Repository svn://svn.zope.org/repos/main/ZODB

Die ZODB-Mailingliste zodb-dev@lists.zope.org, see http://mail.zope.org for subscribing